
Introduction to Professional Web Development in JavaScript

Aug 15, 2019

CONTENTS

1	Contents	1
----------	-----------------	----------

INTRODUCTION

1.1 Why Learn To Code?

How many times do you use a computer in a day? What do you use it for? Maybe you use one to check email and social media, to watch TV, and to even set an alarm for the next day. Computers and technology are *everywhere* in our society.

With the rise of technology and computers, coding has risen as well. At its most basic level, coding is how humans communicate with computers. With code, humans tell computers to complete specific tasks and store specific information. Many would argue that due to the prevalence of computers, learning to code is vital to living in the 21st century. Writing and reading code is becoming a new form of literacy for today's world.

As our needs change, technology changes to meet them. When we needed a way to talk to each other over long distances, we got phones. As our needs to communicate changed, our phones became portable. Then, we gained the ability to use our phones to send quick written messages to each other.

The technical skills required to make the phones of 20 years ago are not the same skills required to make a phone today. A career as a technologist, specifically as a programmer, is one of lifelong learning.

Learning to code is not only valuable and challenging, it is also fun. Every "EUREKA!" moment inspires us to keep going forward and to learn new things. You may find some concepts difficult to understand at first, but these will also be the skills you take the most pride in mastering. While the journey to learning to code is long and winding, it is also rewarding.

From the moment that you write your first line of code, you are a programmer. We hope you enjoy the flight!

1.2 Why Learn JavaScript?

With all the different coding languages in the world, it can be difficult to choose which one to learn first. **JavaScript** is a programming language that has many different applications. Programmers can use JavaScript to make websites, visualize data, and even create art! This class will start with JavaScript for several reasons.

The main reason is that JavaScript has become a key language for web development. Understanding how the internet works and being able to create web applications are important skills in today's landscape. With JavaScript, you can work on your own web applications and support current applications for a company.

Another reason is that JavaScript is very much like other programming languages. Throughout the course and your career, you will hear that once you learn one programming language, it is easier to learn another. Programmers have found this to be true, especially if the new language closely matches the first.

1.3 About LaunchCode Programs

1.3.1 Goals

We want our programs to help you build your problem solving skills and encourage you to learn how to learn. Whether you use the coding skills you gain in this program to get a job as a developer is up to you. However, no matter the path you take after this program, learning how to learn will help you continually adapt to the changing needs of your industry.

To get you ready for a career in technology, our goal is to teach you the skills found in a wide variety of industries.

1.3.2 Blended learning

We only have a short amount of time in class to learn a lot, so using a **blended learning model** helps us make the most of our time in this class. A blended learning model incorporates in-class learning with online materials such as this textbook.

1.3.3 Course activities

We have spent a lot of time creating the course activities to make the most of your in-class and out-of-class time. While there are a lot of different activities we will do in a day, it is important to actively engage with each activity. Skipping the prep work or falling behind on assignments can quickly lead to struggling in the class.

Prep Work

Prep Work is done before each class session and covers the topics that you will learn about that day. In addition to reading, prep work includes small questions that can help you reinforce your understanding of what you have just read. We have found that studying the material before class helps students make the most of their short in-class time.

Exercises

Exercises are small coding problems and are a chance for you to implement what you have just learned. While exercises do not count towards your final grade in the class, it is essential to practice in order to reinforce your understanding of the new concepts.

In-class Time

In class you will meet many fellow students on the same learning journey as you. We encourage students to engage, interact, and encourage each other throughout the class.

In class there will also be an instructor and teaching assistants. This dedicated staff facilitates the activities and provides support to the students.

Large Group Time

During the large group time, the whole class participates in the lesson, led by the instructor. The lesson is not a substitute for doing the prep work before class. It's a time for us to review examples as a group and shore up concepts from the prep work.

Small Group Time

After the large group time, we will break up into small groups, each led by a teaching assistant. During small group time, we will do in-class coding activities called studios. This time is equally as important as large group time because it is time for individual support and a place where you can feel comfortable talking openly about concepts you are struggling with.

Assignments

Assignments are larger projects where you demonstrate what you have learned and challenge yourself. Assignments oftentimes cover multiple lessons.

1.4 Class Platforms

Besides this book, this class uses additional platforms for enrollment, assignments, and grading.

1.4.1 Canvas

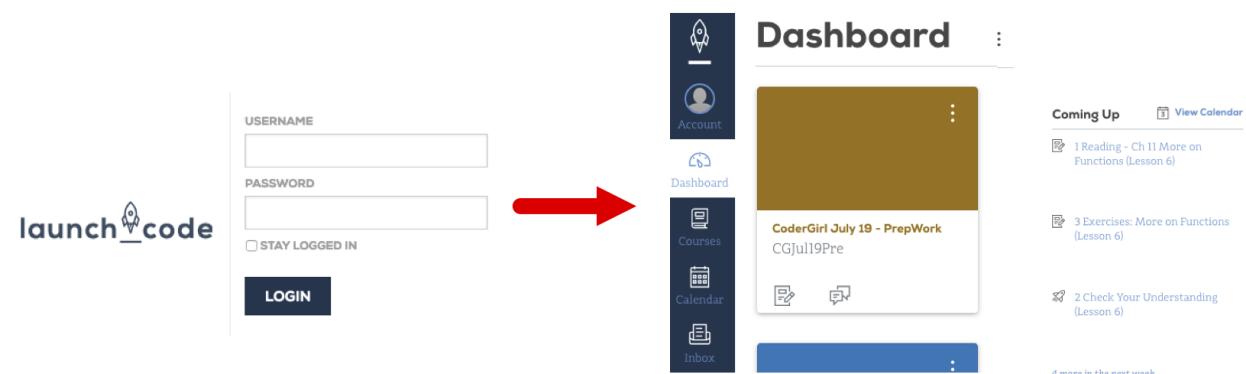
LaunchCode monitors your progress in this class through a management system called *Canvas*. It provides a central location to manage the flow of information, but it does not hold the actual course content. Instead, it links to the lessons you need, and it keeps a record of your completed assignments and scores.

Login to Canvas

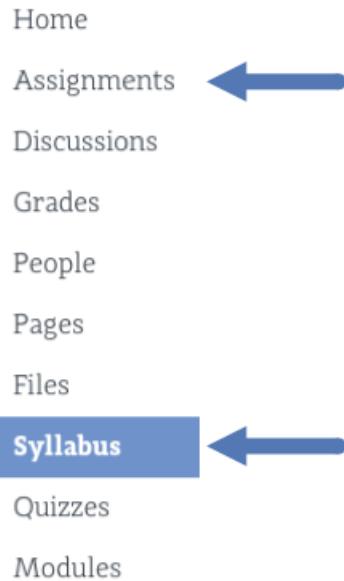
Access Canvas and the course assignments at <https://learn.launchcode.org/>. To login, use your launchcode.org user-name and password, which are the same ones you used to apply for this class.

Canvas Dashboard

After logging in, you will arrive at your *dashboard*, which displays the LaunchCode courses you can access, upcoming due dates, and several menu items.



Clicking on a course title takes you to that class' homepage, which shows upcoming due dates, announcements, general information, and menu options. You will probably use the *Syllabus* and *Assignments* options the most often.



Syllabus Page

The syllabus page provides general information such as a description of the class, the timeline for the course, a calendar, and a To Do list. Scrolling down on the page shows the *Course Summary*, which holds links to individual tasks (reading, quizzes, assignments, etc.).

This page is a good place to answer the questions *What do I need to do next*, and *How can I quickly find and review an old topic*.

Course Summary:

Date	Details	
Sat Jun 15, 2019	Prep Work Quiz	due by 11:59pm
	Prep Work Reading	due by 11:59pm
Mon Jun 17, 2019	1 Reading: Data and Variables (Lesson 1)	due by 1pm
	2 Exercises: Data and Variables (Lesson 1)	due by 1pm
	3 Studio: Data and Variables (Lesson 1)	due by 3pm

Assignments Page

This page sorts required tasks by date or type. A few days before each class session, new tasks will appear on the list. Old content remains active, allowing you to use the links for reference and review.

[SHOW BY DATE](#) [SHOW BY TYPE](#)

▶ **Assignments**

▶ **Prep Work**

▼ **Lesson 1**

1 Reading: Data and Variables (Lesson 1)
 Due Jun 17 at 1pm

2 Exercises: Data and Variables (Lesson 1)
 Due Jun 17 at 1pm

Clicking on a specific title brings up information about that task, including the due date, points possible, instructions, and links.

1 Reading - Ch 7 Stringing Characters Together (Lesson 3)

Due Jun 24 by 12pm Points None

Please read [Chapter 7 - Stringing Characters Together](#)

2 Check Your Understanding - Ch 7 (Lesson 3)

Due Jun 24 at 12pm Points 7 Questions 7
Time Limit None Allowed Attempts Unlimited

Instructions

Please answer these questions after completing: [1Reading - Ch 7 Stringing Characters Together \(Lesson 3\)](#)

[Take the Quiz](#)

Even though much of the course content can be accessed without logging in, the best choice is to begin from within Canvas. That way your progress gets recorded, and your scores will update smoothly as you complete quizzes. Also, submitting files for the larger assignments should only be done through Canvas.

1.4.2 Repl.it

Repl.it is a free online code editor, and it provides a practice space to boost your programming skills.

For this class, repl.it serves two purposes:

1. To provide opportunities to respond to prompts, questions, and “Try It” exercises embedded within the reading. These tasks are neither tracked nor scored.
2. To hold larger exercises and studios that will be checked for accuracy and tracked for completion.

Repl.it Account Creation

Creating a standard repl.it account is covered in *Chapter 2*, and logging in allows students to complete the general practice tasks.

To access the tracked and scored tasks, students must also enroll in a repl.it *classroom*. A link to join the classroom will be provided as part of an early Canvas assignment.

Standard Repl.it

Standard repl.it is an online code editor for various languages. Coders collaborate by sharing repl.it URLs.

Standard repl.it is used for:

1. Publicly sharing code examples and starter code
2. A place to practice new concepts by writing and running code

Repl.it Classroom

Repl.it *Classroom* provides online code editing and submission, but it also allows teachers to post instructions and establish automatic code evaluation.

Assignments in a classroom can kept private from people who are not enrolled, and grades can only be viewed by designated instructors.

Repl.it Classroom is used for:

1. Completing then submitting exercises and getting feedback
2. Completing then submitting studios and assignments for a grade
3. Keeping student submissions visible ONLY to the teachers in the repl.it classroom

Tip

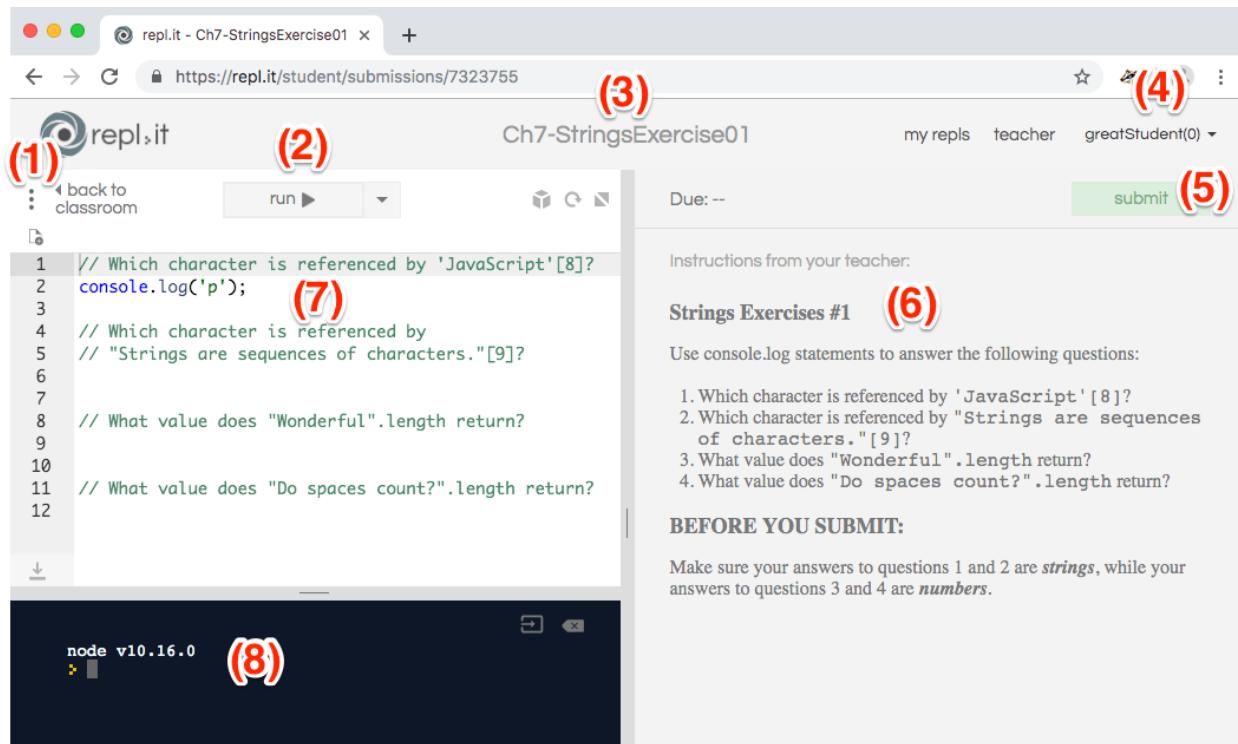
You NEVER have to click save when using standard repl.it or repl.it classroom. Repl.it automatically saves your code on their servers.

However, for repl.it classroom you DO have to click “Submit” to turn in your work.

Repl.it Classroom Workspace

Features to note:

1. **Settings menu:** Allows you to change editor settings like theme and font size.
2. **Run button:** Runs code in the editor panel (7).
3. **Name of assignment**
4. **Repl.it user menu:** Allows you to change account details and change password.
5. **Submit button:**
 - a. Submits your work.
 - b. If the assignment has been setup to be auto graded, you will receive instant feedback.
 - c. If the assignment is NOT auto graded, your TA will need to review and provide feedback.



6. **Instructions:** This describes what you need to code. READ CAREFULLY!
7. **Editor panel:** This is where you write your code.
8. **Console output:** After clicking the run button (2), output and errors will show up here.

Note

Results from work submitted in repl.it classroom, appear in Canvas after being verified.

Remember, Canvas holds student grades and quizzes but NOT the course content. Instead, it provides *links* to the reading and other assignments.

1.5 Using This Book

Throughout this book, you will find a variety of different sections and practice exercises. We are writing this guide to help you make the most of the book.

1.5.1 Concept checks

Many sections end with a “Check Your Understanding”. This section is full of questions for you to double check that you understand the concepts in the reading. Although your score does not count towards your final grade in the class, you should use it to help evaluate your understanding of the main concepts.

1.5.2 Examples

Examples are times when we tie a concept we have just learned to a potential real world application.

The label “Try It” signals an example that includes code you can modify and augment to quickly reinforce what you have just read. Play around with these!

Repl.it

Many examples have links to **Repl.it**. This website allows you to write and run practice code. Repl.it accounts are free, so we encourage you to [sign up for one](#). More details for using Repl.it will be provided when you *write your first program*.

As you explore the prepared examples in this book, feel free to make changes to the code. If you want to save your edits, click the *Fork* button at the top of the workspace, and Repl.it will store a copy of the code in your personal account.

1.5.3 Supplemental Content

Occasionally, you will find a link to “Booster Rocket”. While not required reading, “Booster Rockets” can boost your learning.

1.5.4 JavaScript in Context

Our approach is different from other ways you can learn JavaScript. The book focuses on programming fundamentals. These fundamentals are problem-solving and transferable concepts. While we will cover the exact way to perform certain tasks in JavaScript, we want to remind you that these tasks are relatively common and many programming languages have ways to carry them out.

HOW PROGRAMS WORK

2.1 Introduction

“It’ll take a few moments to get the coordinates from the navicomputer.” - Han Solo

Given a set of inputs, Han’s computer analyzes the data and returns information about safely navigating a hyperspace jump. The computer does this by running a **program**.

At the most basic level, a *program* is a set of instructions that tell a computer or other machine what to do. These instructions consist of a set of commands, calculations, and manipulations that achieve a specific result. However, the computer cannot solve the problem on its own. Someone—a programmer—had to figure out a series of steps for the computer to follow. Also, the programmer had to write these steps in a way the computer can understand.

2.1.1 Algorithms

Imagine following a recipe for baking a batch of cookies. After the list of ingredients comes a series of step-by-step instructions for producing the treats. If you want to make something else, like a cake or a roast, you follow a different set of steps using a different set of ingredients.

An **algorithm** is like a recipe. It is a systematic series of steps that, when followed, produce a specific result to help solve a problem. Programmers design algorithms to solve these small steps in a carefully planned way. The results then get combined to produce a final answer or action.

Let’s take a look at an example of an algorithm—alphabetizing a list of words:

apple, pear, zebra, box, rutabaga, fox, banana, socks, foot

One possible set of steps for solving the task could be:

1. Arrange the words from a - z based only on the first letter:

apple, box, banana, fox, foot, pear, rutabaga, socks, zebra

2. If more than one word starts with ‘a’, rearrange those words based on the second letter. Repeat for the words that start with ‘b’, then ‘c’, etc.:

apple, banana, box, fox, foot, pear, rutabaga, socks, zebra

3. If multiple words start with ‘a’ and have the same second letter, rearrange those words based on the third letter. Repeat for the ‘b’ words, then the ‘c’ words, etc.:

apple, banana, box, foot, fox, pear, rutabaga, socks, zebra

4. If other repeats exist, continue sorting the list by comparing the 4th, 5th, 6th letters (etc.) until all the words are properly arranged.

This is not the ONLY way to solve the task, but it provides a series of steps that can be used in many different situations to organize different lists of words.

Alphabetizing is a process we can teach a computer to do, and the algorithm will complete the process much more rapidly than a human. However, unlike the alphabet song that many of us still sing in our heads when arranging a list of words, programmers must use a different method to train the computer.

2.1.2 Check Your Understanding

Question

Select ALL of the following that can be solved by using an algorithm:

- a. Answering a math problem.
 - b. Sorting numbers in decreasing order.
 - c. Making a peanut butter and jelly sandwich.
 - d. Assigning guests to tables at a wedding reception.
 - e. Creating a grocery list.
 - f. Suggesting new music for a playlist.
 - g. Making cars self-driving.
-

2.2 Programming Languages

“Computer, scan the surface for lifeforms.”

“Hey Siri, what movies are playing nearby?”

Even though today’s tech makes it seem like computers understand spoken language, the devices do not use English, Chinese, Spanish, etc. to carry out their jobs. Instead, programmers must write their instructions in a form that computers understand.

Computers operate using **binary code**, which consists only of 0s and 1s. For example, here is the binary version of the text Hello World:

```
01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111  
01110010 01101100 01100100
```

Each set of 8 digits represents one character in the text.

To make things a little easier, binary data may also be represented as **hexadecimal** values. Here is Hello World expressed in *hex*:

```
48 65 6c 6c 6f 20 57 6f 72 6c 64
```

To run an algorithm, all of the steps must be written in binary or hex so the computer can understand the instructions.

Note

Fortunately, we do not need to worry about binary or hexadecimal code to make our programs work!

2.2.1 Languages

Writing code using only 0s and 1s would be impractical, so many clever individuals designed ways to convert between the text readable by humans and the binary or hexadecimal forms needed by machines.

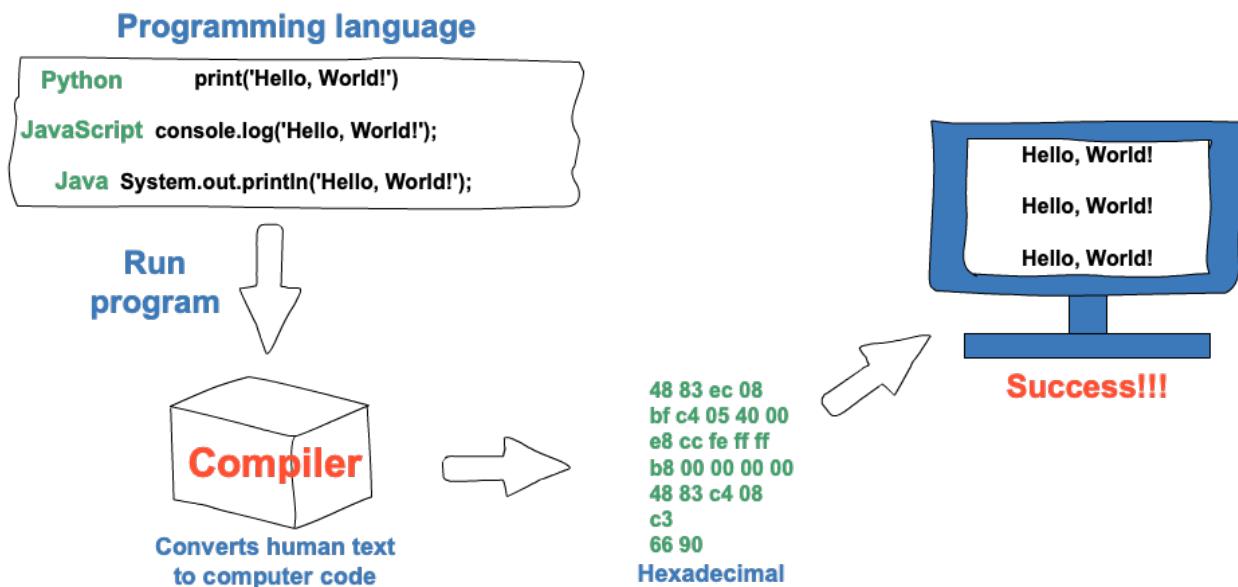
A **programming language** is a set of specific words and rules for teaching a computer how to perform a desired task. Examples of programming languages include Python, JavaScript, Basic, COBOL, C++, C#, Java, and many others.

These *high-level languages* can be written and understood by humans, and each one has its own characteristic vocabulary, style, and syntax.

How Computers Run Programs

Since computers only understand binary code, every programming language includes a **compiler**, which is a special tool that translates a programmer's work into the 0s and 1s that the machines need.

If we want to print `Hello, World!` on the screen, we would write the instructions in our chosen programming language, then select "Run". Our code gets sent to the compiler, which converts our typed commands into something the computer can use. The instructions are then executed by the machine, and we observe the results.



In the example above, the *syntax* for printing `Hello, World!` varies between the Python, JavaScript, and Java languages, but the end result is the same.

2.2.2 How Many Programming Languages Are There?

Ask Google, "How many programming languages are there?" and many results get returned. Even with all these options, there is no specific answer to the question.

There are hundreds, if not thousands, of programming languages available. However, most are either obsolete, impractical, or too specialized to be widely used.

Arguments occur whenever someone makes a top 10 list for programming languages, but regardless of the opinions, one fact remains. Once you learn one language, learning the next is much, much easier. Adding a third becomes child's play.

The reason for this is that thinking like a coder does not change with the language. Your logic, reasoning, and problem solving skills apply just as well for JavaScript as they do for Python, Swift and C#. To display text on the screen in Python, we use `print()`, for JavaScript we use `console.log();`, for C# the command is `Console.WriteLine();`. The *syntax* for each language varies, but the results are identical.

2.3 The JavaScript Language

JavaScript is one of many programming languages, each of which serves different purposes. Programmers mainly use JavaScript for web development, and it is currently the most popular language that runs inside a web browser. *Running inside the browser* means that the code loads at the same time as a web page and can modify the content. JavaScript can add or remove text, change colors, produce animations, and react to mouse and keyboard clicks. This makes the web page *dynamic*—it responds to user actions in real time, and changes occur without having to refresh the page.

This cool feature allows immediate updates to your profile when you post a status, change the color of the “Submit” button after you complete a form, generate a map when you request directions, or create a sparkly trail that follows the mouse pointer. All of the changes to what you see on the web page are part of **front end** development. They are present on your computer.

Back end development involves passing data between web pages and servers. JavaScript can be used for back end development, but other languages like Java and C# are industry standards. When you fill out a form online and click “Submit”, back end code transfers the information you entered to the company that posted the form. Your information now exists on the company’s servers.

The ins and outs of how the internet works will be covered throughout this book. While important to understand why we are learning JavaScript, we won’t quiz you on servers and front end development now!

2.3.1 Front End vs. Back End Changes

2.4 Your First Program

We haven’t learned how to code yet, but we can still write and run our first program. This exercise asks you to create and run small amounts of code, and it reinforces the LaunchCode principle of learning by doing.

We have used the phrase `Hello, World` as an example throughout this chapter because it represents the traditional first program for a new coder. Printing a single message is one of the simplest tasks a program can carry out.

`Hello, World` will be your first program as well. Welcome to the club!

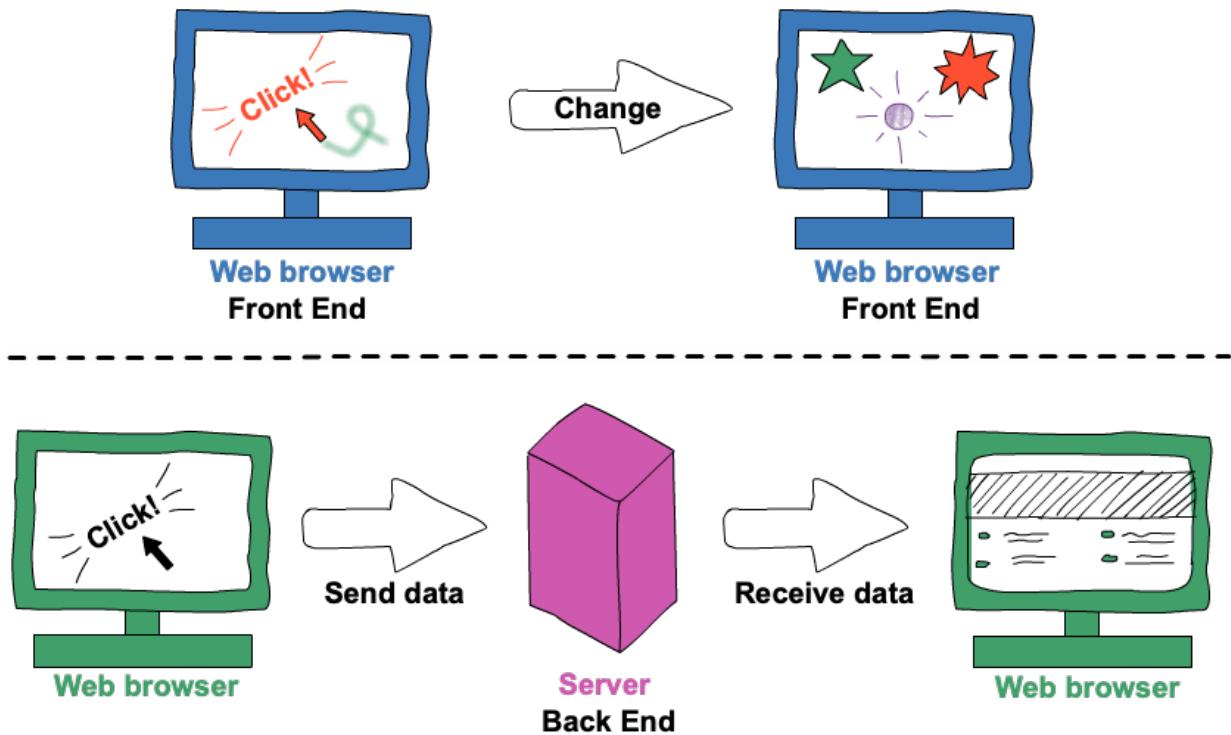
2.4.1 Create a Repl.it Account

Throughout this book, you will need to access a code editor to complete practice problems, exercises, studios, and assignments. If you have not already done so, create a new account with [Repl.it](#). The site provides a free space to practice coding.

After you sign in, you will see your *dashboard*, which displays any saved folders or projects. Since you are just starting out, your dashboard will be empty.

Warning

These examples are for standard repl.it use. See *repl.it classroom instructions* for instructions on using repl.it when working on repl.it classroom assignments.



Click on *new repl* to begin a new project. Scroll through the options and select “Node.js”. Next, name your project and click “Create Repl”.

The Repl.it Workspace

Before you dive into your Hello, World! program, let’s take a look at how to use Repl.it. The workspace consists of three main panels and several menu functions.

Features to note:

1. **File panel and menus:** Allows you to add extensions, update settings, and add, open, or delete files.
2. **Editor panel:** Your code goes here. Click on a file to open it in the editor. For most new projects, an `index` file will be created and opened by default.
3. **Console panel:** Any output produced by your code will appear in this panel. The console also displays error messages, test results, and other information.
4. **Fork button:** If you are viewing someone else’s project, you can *fork* the content (4) and store a copy of that project to your own account. This allows you to edit the files without changing the originals, and it lets you use other programmers’ work (with permission) to enhance your own.
5. **Run button:** Executes any code written in the `index` file.
6. **Managing projects:** When logged into Repl.it, you can create a new project, view saved projects, or share your projects.

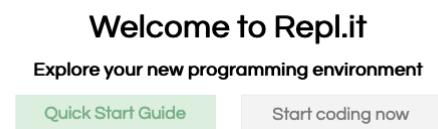
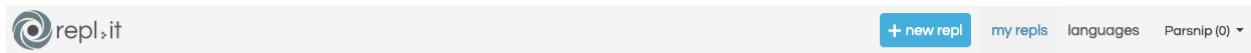
Note

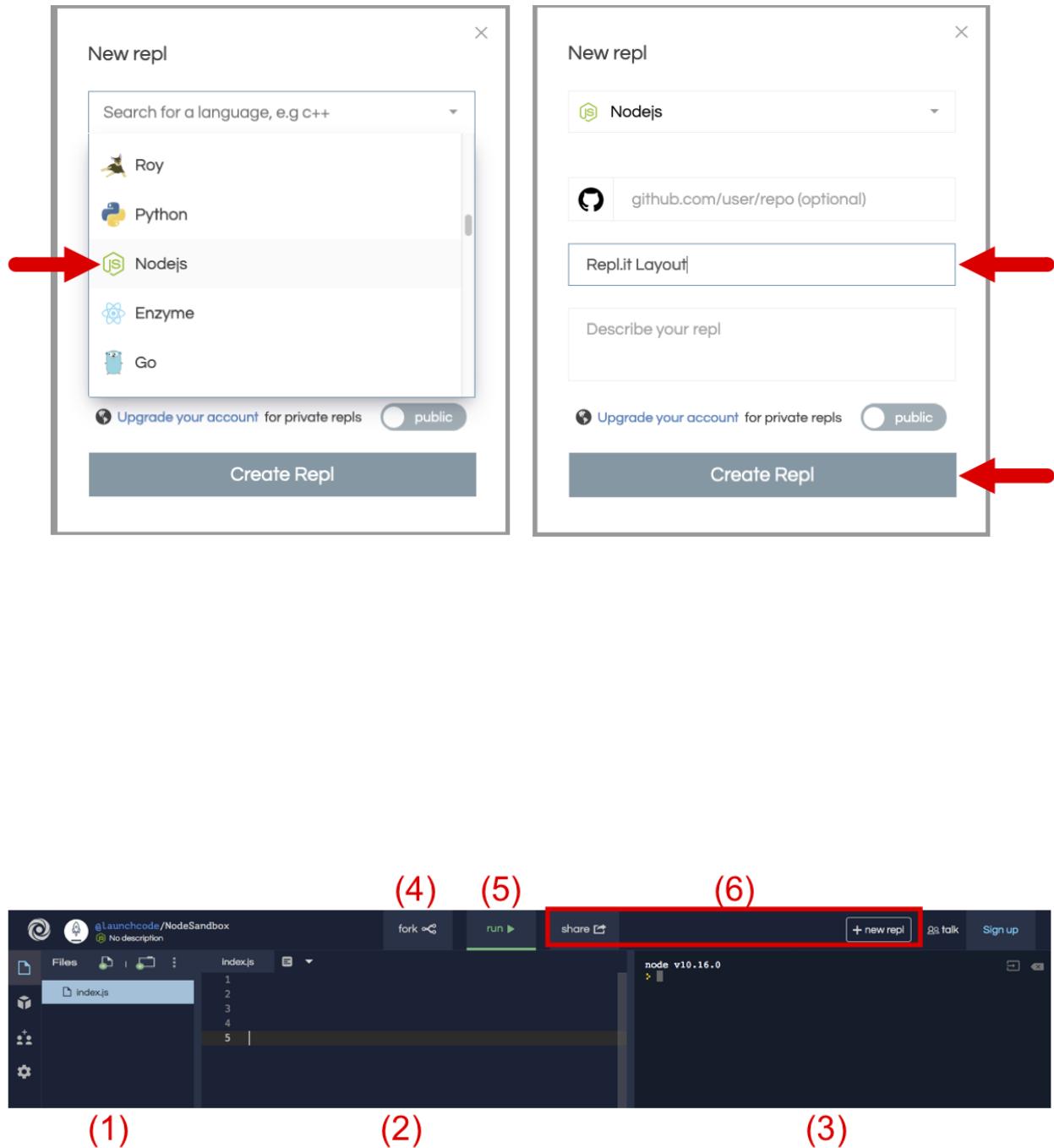


I'm a teacher or [log in](#)

[Sign up](#)

By continuing, you agree to Repl.it's [Terms of Service](#) and [Privacy Policy](#), and to receiving emails with updates.





The workspace shown above uses the “dark” theme (light text on a black background). If you prefer the reverse (dark text on a white background), click the gear icon and select the “light” theme.

2.4.2 Begin Your Coding Journey

Follow this [Hello World link](#) to open a prepared workspace for your first program.

On line 2 of the editor, type:

```
console.log("Hello, World!");
```

When you finish typing, click the green “Run” button and observe the output.

Warning

Do NOT just copy/paste the code. You will learn best by typing, trying, changing, and fixing.

If you typed correctly, you saw the output `Hello, World!` If you omitted or mistyped any characters, then you either saw a misspelled output or an error message with some tips on what might have gone wrong. Do not worry if you make mistakes! These experiences still teach you something. Fix any errors and try again.

Now Play

Once you print `Hello, World!` successfully, go back and play around with the code. Make a change, click “Run”, and see what happens. Try to:

1. Change the message that is printed.
2. Figure out what the parentheses do. Will the code work without them?
3. Remove one or both quotation marks. Do we need to include both opening and closing quote marks? Is there a difference between using a single or a double quote (`'` vs. `"`)?
4. Remove the semi-colon, `;`.
5. Print a number. (Bonus: Print two numbers added together).
6. Print multiple messages one after the other.
7. Print two messages on the same line.
8. Print a message that contains quote marks, such as `Quoth the Raven "Nevermore".`
9. Other. You choose!

Spend a few minutes trying these changes. Do not worry if you miss some of the targets. Learning comes through experience, and you WILL learn all the details behind `console.log` soon.

Once you finish practicing (and hopefully making some mistakes), you will have a pretty good idea of how the `console.log` function in JavaScript works.

Try It

On paper (or in a document on your computer), write one or two sentences about `console.log`. You should provide more detail than, “It prints things.”

2.4.3 Check Your Understanding

Question

Which of the following correctly prints Coding Rocks? There may be more than one valid option.

- a. `console.log(Coding Rocks)`
 - b. `console.log(Coding Rocks);`
 - c. `console.log('Coding Rocks')`
 - d. `console.log("Coding Rocks");`
 - e. `console.log("Coding Rocks");`
-

HOW TO WRITE CODE

3.1 What is Code?

Computers are dumb, understanding not an ounce of context or intended meaning. They react mechanically to the instructions we give them, and they cannot deviate from the steps we tell them to follow.

If our instructions are even the slightest bit off, computers cannot consider the error and adjust accordingly. Instead, they come to a grinding halt.

So how do we give computers instructions in the first place? The answer is to create **code**, which is a set of instructions for a computer to follow.

3.1.1 What Code Can Do

Here is a short list of SOME of the tasks we can carry out with code:

1. Interact with users. Through code, we can ask a user questions, store the answers, and respond by changing what is on the screen.
2. Interact with other systems. Through code, we can interact with resources that are outside of our program. For example, we can read data in from a file on our computer, or we can ask a server on the other side of the planet for information.
3. Repeat tedious tasks. Have a few thousand emails to send? Need to spellcheck several thousand words? You can do these things with just a handful of code.
4. Reuse useful code snippets. Rather than copy/paste the same lines of code in multiple places, we can assign a name to that code. This allows us to use it wherever we like by simply referring to its name.
5. Decide what to do based on the current situation. When we write code, we often need to carry out one task under a specific set of circumstances, but another task if the circumstances differ. We can write code to decide which action to take.

Of course, in order to work, code needs to follow a specific set of rules.

3.2 Syntax Rules

At their core, programming languages are collections of rules that allow us to tell a computer what to do. Actions like *Repeat ____ 25 times*, *Prompt the user for a password*, or *Display text on the screen* can be done with any language. However, each one uses different methods to complete the tasks.

Syntax refers to the structure of a language (spoken, programming, or otherwise) and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with proper punctuation.

JavaScript and other languages can only run a program if it is *syntactically correct*, and each language is pretty rigid and unforgiving if you make a mistake. While humans are good at overlooking minor grammar and syntax errors, computers cannot do the same.

For example, the following sentence will send English teachers into fits, *i believe that Catch 22 demon straights a cleer picture of whirled wor too*. Despite being badly written, we can still make some sense out of the words. It might take some re-reading, but eventually we will get the point. Other examples include vanity license plates:

Plate	Meaning
KC ROKS	Kansas City (or Casey?) Rocks
4EVERL8	All parents who ever needed to get the kids to practice

Computers cannot interpret or overlook mistakes like humans. Any syntax errors, no matter how minor, will prevent the code from running. Instead of trying to work around the issue, the program will immediately crash and generate error messages.

3.3 Comments

As programs get bigger and more complicated, they get more difficult to read. Good programmers try to make their code understandable to others, but it is still tricky to look at a large program and figure out what it is doing and why.

Best practice encourages us to add notes to our programs, which clearly explain what the program is doing. These notes are called *comments*.

A **comment** is text within a program intended only for a human reader—it is completely ignored by the compiler. In JavaScript, the // token starts a comment, and the rest of the line gets ignored. For comments that stretch over multiple lines, the text falls between the symbols /* */.

Try It

Experiment by adding and removing comments to the code.

```
1 // This demo shows off comments!
2
3 // console.log("This does not print.");
4
5     "Hello, World!" ↴
6 ↪// Comments do not have to start at the beginning of a line.
7
8 /* Here is how
9    to have
10   multi-line
11  comments. */
12
13     "Comments make your code more readable by others."
```

repl.it

Notice that when you run the program, it still prints the phrase Hello, World, but none of the comments appear. Also notice the blank lines left in the code, which are also ignored by the compiler. Comments and blank lines make your programs much easier for humans to understand. Use them frequently!

3.4 Output With `console.log`

In the *Hello World* section, you experimented with displaying text on the screen. Technically, you sent the words to the **console**, which is a simple window where the user can type commands or view output. We used a print function without explicitly talking about how it works. Let's fix that now.

We *call* the print function using the syntax `console.log()`. When the code runs, we want it to tell the computer, *Please display what is inside the () on the screen*. For us, the words are enough - we want to LOG the text to the CONSOLE. However, the computer only understands binary or hexadecimal instructions. We need the compiler to change the keywords *console* and *log* into a format that the machine understands.

3.4.1 Examples

Open the repl.it link in the example below, and note the difference between the outputs:

```
1      'Hello, JavaScript.'
2      2001
3      "Spot" "the" "difference"
4      "Spot" "the" "difference"
5      'Launch' 'Code'
6      "LaunchCode was founded in" 2013
```

repl.it

Observations line by line:

1. In the line 1, we print some text, which is surrounded by quotes.
2. In the line 2, we print a number. Note the absence of quote marks.
3. In line 3, we use three words, separated by commas, all within the same set of parentheses (). When these three words print, they show up on the same line but separated by spaces.
4. The code in line 4 is just like line 3, only there are no spaces after the commas. How does this affect the output?
5. Line 5 also prints more words, but in this case the code uses + instead of a comma. The result is to print the words without spaces in between.
6. Line 6 prints text and a number with a space in between.

3.4.2 Two Special Characters

One final observation for all of the examples above is that each time we use `console.log`, a **newline** is inserted after the printed content. Think of a newline as the same as hitting the Enter or Return key on your keyboard. The cursor moves to the beginning of the next line.

For the computer, *newline* as an invisible character that is used to tell the machine to move to the next line. It is possible to use this invisible character with the special representation \n.

Try It

Experiment with the newline character.

```
1      "Some Programming Languages:"
2
3      "Python\nJavaScript\nJava\nC#\nSwift"
```

repl.it

In addition to the newline character, there is also a special tab character, \t. Go back to the eight examples above and experiment with using \t and \n.

3.5 Welcome, Novice Coder

Congratulations! You finished the prep work for LC101. You are ready to take the next steps on your learning journey.

Some words of advice:

1. Take advantage of the resources at your disposal—your instructor, TAs, and classmates are here to help.
2. Complete all of the homework and practice tasks. To learn how to code, you need to code.
3. Ask and answer questions.
4. Recognize that making mistakes is part of the learning process.

We applaud your efforts, and we look forward to your success.

DATA AND VARIABLES

4.1 Values and Data Types

Programs may be thought of as being made up of two things:

1. Data
2. Operations that manipulate data

This chapter focuses primarily on the first of these two fundamental components, data.

Data can be stored in a program in a variety of ways. The most basic unit of data is a value.

A **value** is a specific piece of data, such as a word or a number. Some examples are 5, 5.2, and "Hello, World!".

Each value belongs to a category called a **data type**. We will see many different data types throughout the course, the first two of which are the **number** and **string** types. Numeric values such as 4 and 3.3 are numbers. Sequences of characters enclosed in quotes, such as "Hello, World!", are strings, so-called because they contain a string of letters. Strings must be enclosed in either single or double quotes.

If you are not sure what data type a value falls into, precede the value with `typeof`.

Example

```
1   typeof "Hello, World!"  
2   typeof 17  
3   typeof 3.14
```

Console Output

```
string  
number  
number
```

Not surprisingly, JavaScript reports that the data type of "Hello, World!" is `string`, while the data type of both 17 and 3.14 is `number`. Note that some JavaScript environments may print type names and strings with single quotes around them, as in '`string`', '`number`', and '`hello`'.

Note

Notice that `console.log(typeof "Hello, World!");` prints out `string` to the console. The `typeof` keyword is not printed to the console because the statement `typeof "Hello, World!"` is an **expression**. Briefly, expressions are code segments that are reduced to a value. We will learn more about expressions soon.

Note

`typeof` is a JavaScript entity known as an **operator**. It is similar to a function in that it carries out some kind of action, though the syntax is different from that of functions (notice using `typeof` does not require parentheses).

There are data types other than string and number, including object and function, which we will learn about in future chapters.

4.1.1 More On Strings

What about values like `"17"` and `"3.2"`? They look like numbers, but they are in quotation marks like strings.

Run the following code to find out.

Try It!

```
1   typeof "17"
2   typeof "3.2"
```

[repl.it](#)

Question

What is the data type of the values `"17"` and `"3.2"`?

Strings in JavaScript can be enclosed in either single quotes (`'`) or double quotes (`"`).

Example

```
1   typeof 'This is a string'
2   typeof "And so is this"
```

Console Output

```
string
string
```

Double-quoted strings can contain single quotes inside them, as in `"Bruce's beard"`, and single quoted strings can have double quotes inside them, as in `'The knights who say "Ni!"'`.

JavaScript doesn't care whether you use single or double quotes to surround your strings. Once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value.

Warning

If a string contains a single quote (such as `"Bruce's beard"`) then surrounding it with single quotes gives unexpected results.

```
console.log('Bruce's beard');
```

4.1.2 More On Numbers

When you type a large integer value, you might be tempted to use commas between groups of three digits, as in 42,000. This is not a legal integer in JavaScript, but it does mean something else, which is legal:

Example

```
1 42000  
2 42 000
```

Console Output

```
42000  
42 0
```

Well, that's not what we expected at all! Because of the comma, JavaScript chose to treat 42,000 as a *pair* of values. In fact, the `console.log` function can print any number of values as long as you separate them by commas. Notice that the values are separated by spaces when they are displayed.

Example

```
1 42 17 56 34 11 4.35 32  
2 3.4 "hello" 45
```

Console Output

```
42 17 56 34 11 4.35 32  
3.4 'hello' 45
```

Remember not to put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the chapter *How Programs Work*: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intend.

4.1.3 Type Systems

Every programming language has a **type system**, which is the set of rules that determine how the language deals with data of different types. In particular, how values are divided up into different data types is one characteristic of a type system.

In many programming languages, integers and floats are considered to be different data types. For example, in Python 42 is of the `int` data type, while 42.0 is of the `float` data type.

Note

While JavaScript does not distinguish between floats and integers, at times we may wish to do so in our programs. For example, an inventory-tracking program stores items and the number of each item in stock. Since a store cannot have 3.5 shirts in stock, the programmer makes the quantity of each item integer values as opposed to floats.

When discussing the differences between programming languages, the details of type systems are one of the main factors that programmers consider. There are other aspects of type systems beyond just how values are categorized. We will explore these in future lessons.

4.1.4 Check Your Understanding

Question

Which of these is *not* a data type in JavaScript?

1. number
 2. string
 3. letter
 4. object
-

4.2 Type Conversion

Sometimes it is necessary to convert values from one type to another. A common example is when a program receives input from a user or a file. In this situation, numeric data may be passed to the program as strings.

JavaScript provides a few simple functions that will allow us to convert values to different data types. The functions `Number` and `String` will (attempt to) convert their arguments into types `number` and `string`, respectively. We call these **type conversion** functions.

The `Number` function can take a string and turn it into an integer. Let us see this in action:

Example

```
1      "2345"
2  typeof    "2345"
3      17
```

Console Output

```
2345
number
17
```

What happens if we attempt to convert a string to a number, and the string doesn't directly represent a number?

Example

```
"23bottles"
```

Console Output

```
Nan
```

This example shows that a string has to be a syntactically legal number for conversion to go as expected. Examples of such strings are `"34"` or `"-2.5"`. If the value cannot be cleanly converted to a number then `Nan` will be returned, which stands for “not a number.”

Note

`NaN` is a **special value** in JavaScript that represents that state of not being a number. We will learn more about `NaN` and other special values in a later chapter.

The type conversion function `String` turns its argument into a string. Remember that when we print a string, the quotes may be removed. However, if we print the type, we can see that it is definitely 'string'.

Example

```
1      17
2      123.45
3  typeof    123.45
```

Console Output

```
17
123.45
string
```

4.2.1 Check Your Understanding

Question

Which of the following strings result in `NaN` when passed to `Number`? (Feel free to try running each of the conversions.)

1. '3'
 2. 'three'
 3. '3 3'
 4. '33'
-

4.3 Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A **variable** is a name that refers to a value. Recall that a value is a single, specific piece of data, such as a specific number or string. Variables allow us to store values for later use.

A useful visual analogy for how a variable works is that of a label that *points to* a piece of data.

In this figure, the name `programmingLanguage` points to the string value "`JavaScript`". This is more than an analogy, since it also is representative of how a variable and its value are stored in a computer's memory.

With this analogy in mind, let's look at how we can formally create variables in JavaScript.

4.3.1 Declaring and Initializing Variables With `let`

To create a variable in JavaScript, create a new name for the variable and precede it with the keyword `let`:

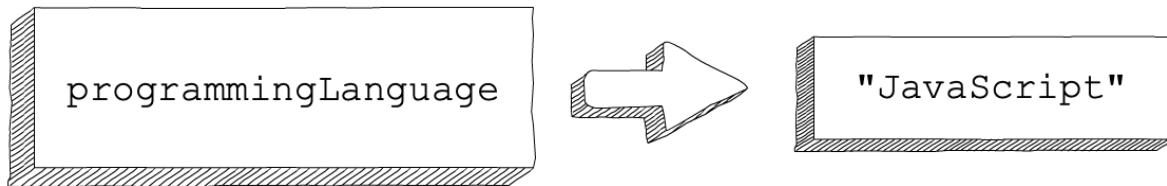


Fig. 1: A variable can be visualized as a label pointing to a specific piece of data.

```
1 let
```

This creates a variable named `programmingLanguage`. The act of creating a variable is referred to as **variable declaration**, or simply **declaration**.

Once a variable has been declared, it may be given a value using an **assignment statement**, which uses `=` to give a variable a value.

```
1 let
2     "JavaScript"
```

The act of assigning a variable a value for the first time is called **initialization**.

The first line creates a variable that does not yet have a value. The variable is a label that does not point to any data.

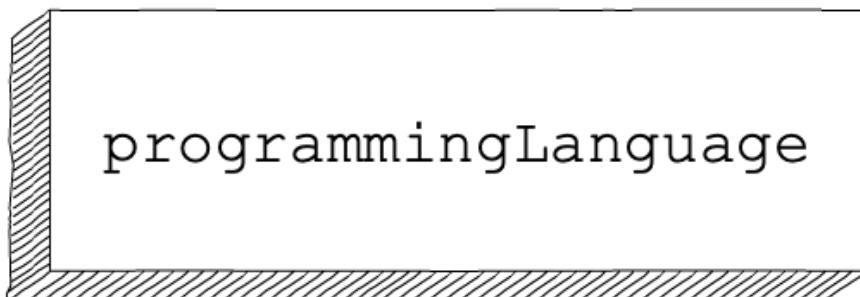


Fig. 2: The result of `let programmingLanguage;`

The second line assigns the variable a value, which connects the name to the given piece of data.

It is possible to declare *and* initialize a variable with a single line of code. This is the most common way to create a variable.

```
1 let
2     "JavaScript"
```

Warning

You will see some programmers use `var` to create a variable in JavaScript, like this:

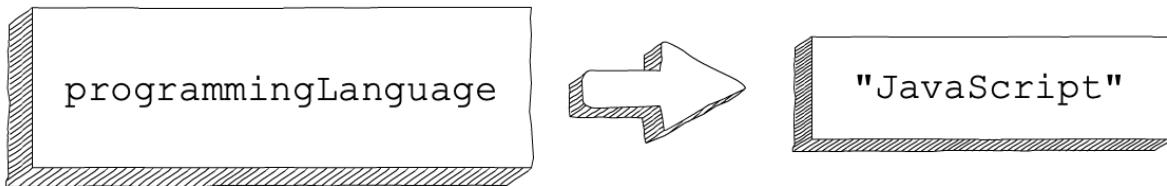


Fig. 3: The result of `programmingLanguage = "JavaScript";`

```
var "JavaScript"
```

While this is valid syntax, you should NOT use `var` to declare a variable. Using `var` is old JavaScript syntax, and it differs from `let` in important ways that we will learn about later. When you see examples using `var`, use `let` instead.

If you're curious, read about [the differences between var and let](#).

To give a variable a value, use the **assignment operator**, `=`. This operator should not be confused with the concept of *equality*, which expresses whether two things are the “same” (we will see later that equality uses the `==` operator). The assignment statement links a *name*, on the left-hand side of the operator, with a *value*, on the right-hand side. This is why you will get an error if you try to run:

```
"JavaScript"
```

An assignment statement must have the name on the left-hand side, and the value on the right-hand side.

Tip

To avoid confusion when reading or writing code, say to yourself:

`programmingLanguage` is assigned '`JavaScript`'

or

`programmingLanguage` gets the value '`JavaScript`'.

Don't say:

`programmingLanguage` equals '`JavaScript`'.

Warning

What if, by mistake, you leave off `let` when declaring a variable?

```
"JavaScript"
```

Contrary to what you might expect, JavaScript will not complain or throw an error. In fact, creating a variable without `let` is valid syntax, but it results in very different behavior. Such a variable will be a **global variable**, which we will discuss later.

The main point to keep in mind for now is that you should *always* use `let` unless you have a specific reason not to do so.

4.3.2 Evaluating Variables

After a variable has been created, it may be used later in a program anywhere a value may be used. For example, `console.log` prints a value, we can also give `console.log` a variable.

Example

These two examples have the exact same same output.

```
"Hello, World!"
```

```
1 let      "Hello, World!"  
2
```

When we refer to a variable name, we are **evaluating** the variable. The effect is just as if the value of the variable is substituted for the variable name in the code when executed.

Example

```
1 let      "What's up, Doc?"  
2 let      17  
3 let      3.14159  
4  
5  
6  
7
```

Console Output

```
What's up, Doc?  
17  
3.14159
```

In each case, the printed result is the value of the variable.

Like values, variables also have types. We determine the type of a variable the same way we determine the type of a value, using `typeof`.

Example

```
1 let      "What's up, Doc?"  
2 let      17  
3 let      3.14159  
4  
5      typeof  
6      typeof  
7      typeof
```

Console Output

```
string  
number  
number
```

The type of a variable is the type of the data it currently refers to.

4.3.3 Reassigning Variables

We use variables in a program to “remember” things, like the current score at the football game. As their name implies, variables can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign it a different value.

To see this, read and then run the following program in a code editor. You’ll notice that we change the value of day three times, and on the third assignment we even give it a value that is of a different data type.

```
1 let      "Thursday"
2
3
4     "Friday"
5
6
7     21
8
```

A great deal of programming involves asking the computer to remember things. For example, we might want to keep track of the number of missed calls on your phone. Each time another call is missed, we can arrange to update a variable so that it will always reflect the correct total of missed calls.

Note

We only use `let` when *declaring* a variable, that is, when we create it. We do NOT use `let` when reassigning the variable to a different value. In fact, doing so will result in an error.

4.3.4 Check Your Understanding

Question

What is printed when the following code executes?

```
1 let      "Thursday"
2     32.5
3     19
4
```

1. Nothing is printed. A runtime error occurs.
 2. Thursday
 3. 32.5
 4. 19
-

Question

How can you determine the type of a variable?

1. Print out the value and determine the data type based on the value printed.
 2. Use `typeof`.
 3. Use it in a known equation and print the result.
 4. Look at the declaration of the variable.
-

Question

Which line is an example of variable initialization? (*Note: only one line is such an example.*)

```
1 let
  2   42
  3     3
```

4.4 More On Variables

The previous section covered creating, evaluating, and reassigning variables. This section will cover some additional, more nuanced topics related to variables.

4.4.1 Creating Constants With `const`

One of the key features of variables that we have discussed so far is their ability to change value. We can create a variable with one value, and then reassign it to another value.

```
1 let           "JavaScript"
  2             "Python"
```

In some situations, we want to create variables that cannot change value. Many programming languages, including JavaScript, provide mechanisms for programmers to make variables that are constant.

For example, suppose that we are writing a to-do list web application, named “Get It Done!” The title of the application might appear in multiple places, such as the title bar and the main page header.

We might store the name of our application in a variable so that it can be referenced anywhere we want to display the application name.

```
let      "Get It Done!"
```

This allows us to simply refer to the `appName` variable any time we want to use it throughout our application. If we change the name of the application, we only have to change one line of code, where the `appName` variable is initialized.

One problem with this approach is that an unwitting programmer might change the value of `appName` later in the code, leading to inconsistent references to the application name. In other words, the title bar and main page header could reference different names.

Using `const` rather than `let` to create a variable ensures that the value of the declared variable cannot be changed.

```
const    "Get It Done!"
```

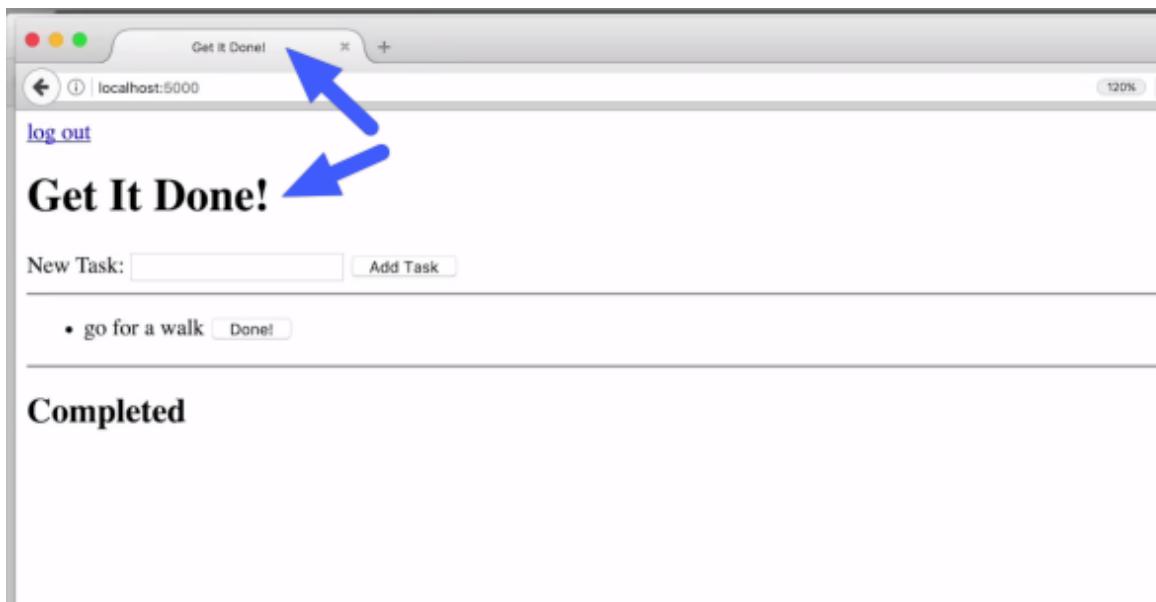


Fig. 4: An example to-do list web application

Such an unchangeable variable is known as a **constant**, since its value is just that.

How does JavaScript prevent a programmer from changing the value of a constant? Let's find out. Try running the following code in an editor. What happens?

Example

```
1 const      "Get It Done"  
2           "Best TODO application Ever!"
```

Console Output

```
TypeError: Assignment to constant variable.
```

As we've seen with other examples—such as trying to declare a variable twice, using incorrect syntax, or failing to enclose strings in quotes—JavaScript prevents undesired code from executing by throwing an error.

4.4.2 Naming Variables

Valid Variable Names

As you may have discovered already, not just any sequence of characters is a valid variable name. For example, if we try to declare a variable with a name containing a space, JavaScript complains.

Example

```
let
```

Console Output

Introduction to Professional Web Development in JavaScript

```
SyntaxError: Unexpected identifier
```

In this case, “identifier” is another term for variable name, so the error message is saying that the variable name is not valid, or is “unexpected”.

JavaScript provides a broad set of rules for naming variables, but there is no reason to go beyond a few easy-to-remember guidelines:

1. Use only the characters 0-9, a-z, A-Z, and underscore. In other words, do not use special characters or whitespace (space, tab, and so on).
2. Do not start a variable name with a number.
3. Avoid starting a variable name with an underscore. Doing so is a convention used by some JavaScript developers to mean something very specific about the variable, and should be avoided.
4. Do not use **keywords**, which are words reserved by JavaScript for use by the language itself. We’ll discuss these in detail in a moment.

Following these guidelines will prevent you from creating illegal variable names. While this is important, we should also strive to create good variable names.

Good Variable Names

Writing good code is about more than writing code that simply works and accomplishes the task at-hand. It is also about writing code that can be read, updated, and maintained as easily as possible. How to write code that achieves these goals is a theme we will return to again and again.

One of the primary ways that code can be written poorly is by using bad variable names. For example, consider the following program. While we haven’t introduced each of the components used here, you should be able to come to a general understanding of the new components.

```
1 let      5
2 const    3.14
3 let      2
4
```

Understanding what this program is trying to do is not obvious, to say the least. The main problem is that the variable names `x`, `y`, and `z` are not descriptive. They don’t tell us anything about what they represent, or how they will be used.

Variable names should be descriptive, providing context about the data they contain and how they will be used.

Let’s look at an improved version this program.

```
1 let      5
2 const    3.14
3 let      2
4
```

With improved variable names, it now becomes clear that the program is calculating the area of a circle of radius 5.

Tip

When considering program readability, think about whether or not your code will make sense to another programmer. It is not enough for code to be readable by only the programmer that originally wrote it.

Camel Case Variable Names

There is one more aspect of naming variables that you should be aware of, and that is conventions used by professional programmers. Conventions are not formal rules, but are informal practices adopted by a group.

Example

In the United States, it is common for two people to greet each other with a handshake. In other countries and cultures, such as some in east Asia, the conventional greeting is to bow.

Failing to follow a social convention is not a violation of the law, but is considered impolite nonetheless. It is a signal that you are not part of the group, or do not respect its norms.

There are a variety of types of conventions used by different groups of programmers. One common type of convention is that programmers that specialize in a specific language will adopt certain variable naming practices.

In JavaScript, most programmers use the **camel case** style, which stipulates that variable names consist of names or phrases that:

- are joined together to omit spaces,
- start with a lowercase letter, and
- capitalize each internal word.

In the example from the previous section, the descriptor “area of circle” became the variable name `areaOfCircle`. This convention is called camel case because the capitalization of internal words is reminiscent of a camel’s humps. Another another common name for this convention is **lower camel case**, since names start with a lowercase letter.

Note

Different programming languages often have different variable-naming conventions. For example, in Python the convention is to use all lowercase letters and separate words with underscores, as in `area_of_circle`.

We will use the lower camel case convention throughout this course, and strongly encourage you to do so as well.

4.4.3 Keywords

Our last note on naming variables has to do with a collection of words that are reserved for use by the JavaScript language itself. Such words are called **keywords**, or **reserved words**.

Any word that is formally part of the JavaScript language syntax is a keyword. So far, we have seen only four keywords: `let`, `const`, `var`, and `typeof`.

Warning

While `console` and `console.log` may seem like keywords, they are actually slightly different things. They are entities (an object and a function, respectively) that are available by default in most JavaScript environments.

Attempting to use a keyword for anything other than it’s intended use will result in an error. To see this, let’s try to name a variable `const`.

Example

```
let const
```

Console Output

```
let const
^^^^^
SyntaxError: Unexpected token const
```

Tip

Most code editors will highlight keywords in a different color than variables or other parts of your code. This serves as a visual cue that a given word is a keyword, and can help prevent mistakes.

We will not provide the full list of keywords at this time, but rather point them out as we learn about each of them. If you are curious, the [full list is available at MDN](#).

4.4.4 Check Your Understanding

Question

Which is the best keyword for declaring a variable in most situations?

1. var
 2. let
 3. const
 4. (no keyword)
-

4.5 Expressions and Evaluation

An **expression** is a combination of values, variables, operators, and calls to functions. An expression can be thought of as a formula that is made up of multiple pieces.

The *evaluation* of an expression produces a value, known as the **return value**. We say that an expression **returns** a value.

Expressions need to be evaluated when the code executes in order to determine the return value, or specific piece of data that should be used. Evaluation is the process of computing the return value.

If you ask JavaScript to print an expression using `console.log`, the interpreter **evaluates** the expression and displays the result.

Example

```
1 1
```

Console Output

```
2
```

This code prints not `1 + 1` but rather the *result* of calculating `1 + 1`. In other words, `console.log(1 + 1)` prints the value 2. This is what we would expect.

Since evaluating an expression produces a value, expressions can appear on the right-hand side of assignment statements.

Example

```
1 let      1  2
2
```

Console Output

```
3
```

The value of the variable `sum` is the result of evaluating the expression `1 + 2`, so the value 3 is printed.

A value all by itself is a simple expression, and so is a variable. Evaluating a variable gives the value that the variable refers to. This means that line 2 of the example above also contains the simple expression `sum`.

4.6 Operations

4.6.1 Operators and Operands

Now that we can store data in variables, let's explore how we can generate new data from existing data.

An **operator** is one or more characters that represents a computation like addition, multiplication, or division. The values an operator works on are called **operands**.

The following are all legal JavaScript expressions whose meaning is more or less clear:

- `20 + 32`
- `hour - 1`
- `hour * 60 + minute`
- `minute / 60`
- `5 ** 2`
- `(5 + 9) * (15 - 7)`

For example, in the calculation `20 + 32`, the operator is `+` and the operands are 20 and 32.

The symbols `+` and `-`, and the use of parentheses for grouping, mean in JavaScript what they mean in mathematics. The asterisk (`*`) is the symbol for multiplication, and `**` is the symbol for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect.

Example

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

Console Output

```
5
-1
6
8
9
```

We use the same terminology as before, stating that `2 + 3` **returns** the value 5.

When a variable name appears in the place of an operand, it is replaced with the value that it refers to before the operation is performed. For example, suppose that we wanted to convert 645 minutes into hours. Division is denoted by the operator `/`.

Example

```
1 let      645
2 let      60
3
```

Console Output

```
10.75
```

In summary, operators and operands can be combined to create expressions that are evaluated upon execution. Let's discuss some specific types of operators

4.6.2 Arithmetic Operators

Some of most commonly-used operators are the **arithmetic operators**, which carry out basic mathematical operations. These behave exactly as you are used to, though the modulus operator (`%`) may be new to you.

Table 1: Arithmetic operators

Operator	Description	Example
Addition (+)	Adds the two operands	$2 + 3$ returns 5
Subtraction (-)	Subtracts the two operands	$2 - 3$ returns -1
Multiplication (*)	Multiplies the two operands	$2 * 3$ returns 6
Division (/)	Divides the first operand by the second	$6 / 2$ returns 3
Modulus (%)	Aka the remainder operator. Returns the integer remainder of dividing the two operands.	$7 \% 5$ returns 2
Exponentiation (**)	Calculates the base (first operand) to the exponent (second operand) power, that is, $\text{base}^{\text{exponent}}$	$3 ** 2$ returns 9 $5 ** -1$ returns 0.2
Increment (++)	Adds one to its operand. If used before the operand ($++x$), returns the value of its operand after adding one; if used after the operand ($x++$), returns the value of its operand before adding one.	If x is 2, then $++x$ sets x to 3 and returns 3, whereas $x++$ returns 2 and, only then, sets x to 3
Decrement (--)	Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 2, then $--x$ sets x to 1 and returns 1, whereas $x--$ returns 2 and, only then, sets x to 1

While the modulus operator (%) is common in programming, it is not used much outside of programming. Let's explore how it works with a few examples.

The % operator returns the *remainder* obtained by carrying out integer division of the first operand by the second operand. Therefore, $5 \% 3$ is 2 because 3 goes into 5 one whole time, with a remainder of 2 left over.

Examples

- $12 \% 4$ is 0, because 4 divides 12 evenly (that is, there is no remainder)
 - $13 \% 7$ is 6
 - $6 \% 2$ is 0
 - $7 \% 2$ is 1
-

The last two examples illustrate a general rule: An integer x is even exactly when $x \% 2$ is 0 and is odd exactly when $x \% 2$ is 1.

Note

The value returned by $a \% b$ will be in the range from 0 to b (not including b).

Tip

If remainders and the modulus operator seem tricky to you, we recommend getting additional practice at [Khan Academy](#).

4.6.3 Order of Operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. JavaScript follows the same precedence rules for its arithmetic operators that mathematics does.

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3 - 1)$ is 4, and $(1 + 1) ** (5 - 2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(\text{minute} * 100) / 60$, even though it doesn't change the result.
2. Exponentiation has the next highest precedence, so $2 ** 1 + 1$ is 3 and not 4, and $3 * 1 ** 3$ is 3 and not 27. Can you explain why?
3. Multiplication, division, and modulus operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So $2 * 3 - 1$ yields 5 rather than 4, and $5 - 2 * 2$ is 1, not 6.
4. Operators with the *same* precedence are evaluated from left-to-right. So in the expression $6 - 3 + 2$, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been $6 - (3 + 2)$, which is 1.

Tip

The acronym PEMDAS can be used to remember order of operations:

P = parentheses

E = exponentiation

M = multiplication

D = division

A = addition

S = subtraction

Note

Due to an historical quirk, an exception to the left-to-right rule is the exponentiation operator `**`. A useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

```
1 // the right-most ** operator is applied first
2   2   3   2
3
4 // use parentheses to force the order you want
5   2   3   2
```

Console Output

512
64

4.6.4 Check Your Understanding

Question

What is the value of the following expression?

16	2	5	3	1
----	---	---	---	---

1. 14
 2. 24
 3. 3
 4. 13.66666666666666
-

Question

What is the output of the code below?

1	5	3
---	---	---

Question

What is the value of the following expression?

2	2	3	3
---	---	---	---

1. 768
 2. 128
 3. 12
 4. 256
-

4.7 Other Operators

4.7.1 The String Operator +

So far we have only seen operators that work on operands which are of type `number`, but there are operators that work on other data types as well. In particular, the `+` operator can be used with `string` operands to **concatenate**, or join together two strings.

Example

Introduction to Professional Web Development in JavaScript

"Launch" + "Code" evaluates to "LaunchCode"

Let's compare + used with numbers to + used with strings.

Example

```
1   1   1  
2 "1"  "1"
```

Console Output

```
2  
11
```

This example demonstrates that **the operator + behaves differently based on the data type of its operands**.

Warning

So far we have only seen examples of operators working with data of like type. For the examples `1 + 1` and `"1" + "1"`, both operands are of type `number` and `string`, respectively.

We will explore such “mixed” operations in a later chapter.

4.7.2 Compound Assignment Operators

A common programming task is to update the value of a variable in reference to itself.

Example

```
1 let   1  
2   1  
3  
4
```

Console Output

```
2
```

Line 2 may seem odd to you at first, since it uses the value of the variable `x` to update `x` itself. This technique is not only legal in JavaScript (and programming in general) but is quite common. It essentially says, “update `x` to be one more than its current value.”

This action is so common, in fact, that it has a shorthand operator, `+=`. The following example has the same behavior as the one above.

Example

```
1 let   1  
2   1  
3  
4
```

Console Output

```
2
```

The expression `x += 1` is shorthand for `x = x + 1`.

There is an entire family of such shorthand operators, known as **compound assignment operators**.

Table 2: Compound Assignment Operators

Operator name	Shorthand	Meaning
Addition assignment	<code>a += b</code>	<code>a = a + b</code>
Subtraction assignment	<code>a -= b</code>	<code>a = a - b</code>
Multiplication assignment	<code>a *= b</code>	<code>a = a * b</code>
Division assignment	<code>a /= b</code>	<code>a = a / b</code>

4.8 Input with readline-sync

`console.log` works fine for printing static (unchanging) messages to the screen. If we wanted to print a phrase greeting a specific user, then `console.log("Hello, Dave.");` would be OK as long as Dave is the actual user.

What if we want to greet someone else? We could change the string inside the `()` to be `'Hello, Sarah'` or `'Hello, Elastigirl'` or any other name we need. However, this is inefficient. Also, what if we do not know the name of the user beforehand? We need to make our code more general and able to respond to different conditions.

It would be great if we could ask the user to enter a name, store that string in a variable, and then print a personalized greeting using `console.log`. Variables to the rescue!

4.8.1 Requesting Data

To personalize the greeting, we have to get **input** from the user. This involves displaying a **prompt** on the screen (e.g. “Please enter a number:”), and then waiting for the user to respond. Whatever information the user enters gets stored for later use.

As we saw earlier, each programming language has its own way of accomplishing the same task. For example, the Python syntax is `input("Please enter your name: ")`, while C# uses `Console.ReadLine();`.

JavaScript also has a built-in module for collecting data from the user, called `readline-sync`. Unfortunately, using this module requires more than a single line of code.

4.8.2 Syntax

Gathering input from the user requires the following setup:

```
1 const           'readline-sync'
2
3 let            "Question text... "
```

There is a lot going on here behind the scenes, but for now you should follow this bit of wisdom:

I turn the key, and it goes.

Most of us do not need to know all the details about how cars, phones, or microwave ovens work. We just know enough to interact with them in our day to day lives. Similarly, we do not need to understand how `readline-sync` works at this time. We just need to know enough to collect information from a user.

As you move through the course, you WILL learn about all of the pieces that fit together to make this process work. For now, here is a brief overview.

Load the Module

In line 1, `const input = require('readline-sync')` pulls in all the functions that allow us to get data from the user and assigns them to the variable `input`.

Recall that `const` ensures that `input` cannot be changed.

How to Prompt the User

To display a prompt and wait for a response, we use the following syntax: `let info = input.question("Question text... ");`

When JavaScript evaluates the expression, it follows the instructions:

1. Display `Question text` on the screen.
2. Wait for the user to respond.
3. Store the data in the variable `info`.

For our greeting program, we would code `let name = input.question("Enter your name: ");`. The user enters a name and presses the Return or Enter key. When this happens, any text entered is collected by the `input` function and stored in `name`.

Try It

Let's play around with the `input` statement. Open the repl.it link below and click the "Run" button.

```
1 const          'readline-sync'  
2  
3 let           "Enter your name: "
```

[repl.it](#)

Note that after entering a name, the program does not actually DO anything with the information. If we want to print the data as part of a message, we need to put `name` inside a `console.log` statement.

After line 3, add `console.log("Hello, " + name + "!");`, then run the code several times, trying different responses to the input prompt.

By storing the user's name inside `name`, we gain the ability to hold onto the data and use it when and where we see fit.

Try adding another `+ name` term inside the `console.log` statement and see what happens. Next, add code to prompt the user for a second name. Store the response in `otherName`, then print both names using `console.log`.

Try It

Update your code to request a user's first and last name, then print an output that looks like:

```
First name: Elite
Last name: Coder
Last, First: Coder, Elite
```

4.8.3 Critical Input Detail

There is one very important quirk about the input function that we need to remember. Given `console.log(7 + 2);`, the output would be 9.

Now explore the following code, which prompts the user for two numbers and then prints their sum:

```
1 const readlineSync = require('readline-sync')
2
3 let num1 = readlineSync.question("Enter a number: ")
4 let num2 = readlineSync.question("Enter another number: ")
```

repl.it

Run the program, enter your choice of numbers, and examine the output. Do you see what you expected?

If we enter 7 and 2, we expect an output of 9. We do NOT expect 72, but that is the result printed. What gives?!?!

The quirk with the `input` function is that it *treats all entries as strings*, so numbers get concatenated rather than added. Just like “Hello,” + “World” outputs as Hello, World, “7” + “2” outputs as 72.

JavaScript treats input entries as strings!

If we want our program to perform math on the entered numbers, we must *use type conversion* to change the string values into numbers.

Try It

1. Use `Number` to convert `num1` and `num2` from strings to numbers. Run the program and examine the result.
 2. Instead of using two steps to assign `num1` and then convert it, combine the steps in line 3. Place `input.question("Enter a number: ")` inside the `Number` function. Run the program and examine the result.
 3. Repeat step 2 for `num2`
 4. What happens if a user enters `Hi` instead of a number?
-

4.8.4 Check Your Understanding

Question

What is printed when the following program runs?

```
1 const readlineSync = require('readline-sync')
2
3 let age = readlineSync.question("Please enter your age: ")
4 // The user enters 25.
```

(continues on next page)

(continued from previous page)

5
6

typeof

1. string
 2. number
 3. info
 4. 25
-

4.9 Exercises: Data and Variables

Exercises appear regularly in the book. Just like the concept checks, these exercises will check your understanding of the topics in this chapter. They also provide good practice for the new skills.

Unlike the concept checks, you will need a code editor to complete the exercises. Fortunately, you *created a free account* on Repl.it as part of the prep work.

Open [this link](#) to the repl.it file prepared for this task, then follow the instructions below.

4.9.1 Practice

Use the information given below about your space shuttle to complete the exercises:

Data	Value
Name of the space shuttle	Determination
Shuttle Speed (mph)	17,500
Distance to Mars (km)	225,000,000
Distance to the Moon (km)	384,400
Miles per kilometer	0.621

1. Declare and assign variables

For each row in the table above, declare and assign variables.

Hint: When declaring and assigning your variables, remember that you will need to use that variable throughout the rest of the exercises. Make sure that you are using the correct data type!

2. Print out the type of each variable

For each variable you declared in the previous section, use the `typeof` operator to print its type to the console.

3. Calculate a space mission!

We need to determine how many days it will take to reach Mars.

1. Create and assign a miles to Mars variable. You can get the miles to Mars by multiplying the distance to Mars in kilometers by the miles per kilometer.
 2. Next, we need a variable to hold the hours it would take to get to Mars. To get the hours, you need to divide the distance to Mars in miles by the shuttle's speed.
 3. Finally, declare a variable and assign it the value of days to Mars. In order to get the days it will take to reach Mars, you need to divide the hours it will take to reach Mars by 24.
4. Print out the results of your calculations

Using the variable from part 3c above, print to the screen a sentence that says "_____ will take _____ days to reach Mars."

Fill in the blanks with the shuttle name and the calculated time.

5. Now calculate a trip to the Moon

Repeat the calculations, but this time determine the number of days it would take to travel to the Moon and print to the screen a sentence that says "_____ will take _____ days to reach the Moon".

4.10 Studio: Data and Variables

In this studio, you are going to write code to display the *very important* **Launch Checklist LC04**.

LC04 displays information related to the space shuttle, astronauts, and rockets before launch.

Click this link to open the starter repl.it file.

4.10.1 Declare and Initialize Variables

Declare and initialize a variable for every data point listed in the table below. Remember to account for the different data types.

Note

For now, use the `string` type for the `date` and `time` values. Later in the class, we will learn other ways to work with date and time in JavaScript.

Variable	Value
<code>date</code>	Monday 2019-03-18
<code>time</code>	10:05:34 AM
<code>astronautCount</code>	7
<code>astronautStatus</code>	ready
<code>averageAstronautMassKg</code>	80.7
<code>crewMassKg</code>	<code>astronautCount * averageAstronautMassKg</code>
<code>fuelMassKg</code>	760,000
<code>shuttleMassKg</code>	74842.31
<code>totalMassKg</code>	<code>crewMassKg + fuelMassKg + shuttleMassKg</code>
<code>fuelTempCelsius</code>	-225
<code>fuelLevel</code>	100%
<code>weatherStatus</code>	clear

4.10.2 Generate the LC04 Form

Display **LC04** to the console using the variables you declared and initialized.

The generated report should look *exactly* like the example below—including spaces and symbols (-, >, and *).

Example Output

Note that your output will change when you assign different values to `astronautCount`, `fuelMassKg`, etc. The point is to AVOID coding specific values into the `console.log` statements. Use your variable names instead.

```
-----  
> LC04 - LAUNCH CHECKLIST  
-----
```

```
Date: Monday 2019-03-18  
Time: 10:05:34 AM
```

```
-----  
> ASTRONAUT INFO  
-----
```

```
* count: 7  
* status: ready
```

```
-----  
> FUEL DATA  
-----
```

```
* Fuel temp celsius: -225 C  
* Fuel level: 100%
```

```
-----  
> WEIGHT DATA  
-----
```

```
* Crew mass: 564.9 kg  
* Fuel mass: 760000 kg  
* Shuttle mass: 74842.31 kg  
* Total mass: 835407.21 kg
```

```
-----  
> FLIGHT PLAN  
-----
```

```
* weather: clear
```

```
-----  
> OVERALL STATUS  
-----
```

```
* Clear for takeoff: YES
```

4.10.3 Show Off Your Code

When finished, show your code to your TA so they can verify your work.

If you do not finish before the end of class, login to Repl.it and save your work. Complete the studio at home, then copy the URL for your project and submit it to your TA.

4.10.4 Bonus Mission

Use `readline-sync` to prompt the user to enter the value for `astronautCount`.

The values printed for `astronautCount`, `crewMassKg`, and `totalMassKg` should change based on the number of astronauts on the shuttle. (Don't forget to convert the input value from a string to a number).

MAKING DECISIONS WITH CONDITIONALS

5.1 Booleans

One of the core features of any programming language is the ability to conditionally execute a segment of code. This means that a program will run a segment of code *only if* a given condition is met.

Example

Consider a banking application that can remind you when a bill is due. The application will notify you that a bill is due soon, but *only if* the bill has not already been paid.

The condition for the above example is: Send a notification of an upcoming bill only if the statement “the bill is unpaid” is true. In order to state something like this in JavaScript, we need to understand how programming languages represent true and false.

5.1.1 Boolean Values

The JavaScript data type for storing true and false values is `boolean`, named after the British mathematician George Boole.

Fun Fact

George Boole created [Boolean Algebra](#), which is the basis of all modern computer arithmetic.

There are only two **boolean values**—`true` and `false`. JavaScript is case-sensitive, so `True` and `False` are *not* valid boolean values.

Example

```
1      true
2      typeof true
3      typeof false
```

Console Output

```
true
boolean
boolean
```

The values `true` and `false` are *not* strings. If you use quotes to surround booleans ("`true`" and "`false`"), those values become strings.

Example

```
1     typeof true
2     typeof "true"
```

Console Output

```
boolean
string
```

5.1.2 Boolean Conversion

As with the number and string data types, the boolean type also has a conversion function, `Boolean`. It works similarly to the `Number` and `String` functions, attempting to convert a non-boolean value to a boolean.

Try It!

Explore how `Boolean` converts various non-boolean values.

```
1     "true"
2     "TRUE"
3     0
4     1
5     ''
6     'LaunchCode'
```

[repl.it](#)

Question

Under which conditions does `Boolean` convert a string to `true`?

1. Only when the string is "`true`".
 2. Whenever the string contains any non-whitespace character.
 3. Whenever the string is non-empty.
 4. Never. It converts all strings to `false`.
-

5.1.3 Boolean Expressions

A **boolean expression** is an expression that evaluates to either `true` or `false`. The equality operator, `==`, compares two values and returns `true` or `false` depending on whether the values are equal.

Example

```

1      5   5
2      5   6
  
```

Console Output

```

true
false
  
```

In the first statement, the two operands are equal, so the expression evaluates to `true`. In the second statement, 5 is not equal to 6, so we get `false`.

We can also use `==` to see that `true` and "`true`" are not equal.

Example

```

true      "true"
  
```

Console Output

```

false
  
```

Comparison Operators

The `==` operator is one of six common **comparison operators**.

Table 1: Comparison Operators

Operator	Description	Examples Returning <code>true</code>	Examples Returning <code>false</code>
Equal (<code>==</code>)	Returns <code>true</code> if the two operands are equal, and <code>false</code> otherwise.	<code>7 == 7</code> <code>"dog" == "dog"</code>	<code>7 == 5</code> <code>"dog" == "cat"</code>
Not equal (<code>!=</code>)	Returns <code>true</code> if the two operands are not equal, and <code>false</code> otherwise.	<code>7 != 5</code> <code>"dog" != "cat"</code>	<code>7 != 7</code> <code>"dog" != "dog"</code>
Greater than (<code>></code>)	Returns <code>true</code> if the left-hand operand is greater than the right-hand operand, and <code>false</code> otherwise.	<code>7 > 5</code> <code>'b' > 'a'</code>	<code>5 > 7</code> <code>'a' > 'b'</code>
Less than (<code><</code>)	Returns <code>true</code> if the left-hand operand is less than the right-hand operand, and <code>false</code> otherwise.	<code>5 < 7</code> <code>'a' < 'b'</code>	<code>7 < 5</code> <code>'b' < 'a'</code>
Greater than or equal (<code>>=</code>)	Returns <code>true</code> if the left-hand operand is greater than or equal to the right-hand operand, and <code>false</code> otherwise.	<code>7 >= 5</code> <code>7 >= 7</code> <code>'b' >= 'a'</code> <code>'b' >= 'b'</code>	<code>5 >= 7</code> <code>'a' >= 'b'</code>
Less than or equal (<code><=</code>)	Returns <code>true</code> if the left-hand operand is less than or equal to the right-hand operand, and <code>false</code> otherwise.	<code>5 <= 7</code> <code>5 <= 5</code> <code>'a' <= 'b'</code> <code>'a' <= 'a'</code>	<code>7 <= 5</code> <code>'b' <= 'a'</code>

Although these operations are probably familiar, the JavaScript symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an *assignment* operator and `==` is a *comparison* operator. Also note that `=<` and `=>` are not recognized operators.

An equality test is *symmetric*, meaning that we can swap the places of the operands and the result is the same. For a variable `a`, if `a == 7` is `true` then `7 == a` is also `true`. However, an assignment statement is not symmetric: `a = 7` is legal while `7 = a` is not.

Warning

If you explore the equality operator in more depth, you will find some surprises. For example, the following comparisons return `true`:

- `7 == "7"`
- `0 == false`
- `0 == ''`

We will explore the nuances of `==` in the upcoming section *Equality*, and introduce two new operators, `====` and `!==`, that will align more closely with our intuitive notion of equality.

5.1.4 Check Your Understanding

Question

Which of the following is a Boolean expression? Select all that apply.

1. `3 == 4`
 2. `3 + 4`
 3. `3 + 4 === 7`
 4. `"false"`
-

5.2 Equality

5.2.1 Loose Equality With `==`

In the section *Booleans*, we learned about the comparison operators `==` and `!=`, which test whether two values are equal or not equal, respectively. However, there are some quirks with using the `==` operator, which occur when we use `==` to compare different data types.

Example

```
1   7      "7"
2   0      false
3   0      ''
```

Console Output

```
true
true
true
```

In order to properly make a comparison, the two operands must be the same type. If the two operands to `==` are of different data types, JavaScript will implicitly convert the operands so that the values are of the same data type before comparing the two. For this reason, the `==` operator is often said to measure **loose equality**.

Type conversions with `==` are carried out according to a [complex set of rules](#), and while many of these conversions make some sense, others do not.

For example, `Number("7")` returns `7`, so it makes some sense that `7 == "7"` returns `true`. However, the following example leaves us scratching our heads.

Example

```
1   '0'      0
2   0      ''
3   '0'      ''
```

Console Output

```
true
true
false
```

The `==` operator is **non-transitive**. We think of equality as being transitive; for example, if A and B are equal and B and C are equal, then A and C are also equal. However, the example above demonstrates that that is *not* the case for the `==` operator.

Since `==` does not follow rules that we typically associate with equality, unexpected results may occur if `==` is used in a program. Thankfully, JavaScript provides another operator that returns more predictable results.

5.2.2 Strict Equality With `===`

The operator `===` compares two operands *without* converting their data types. In other words, if `a` and `b` are of different data types (say, `a` is a string and `b` is a number) then `a === b` will always be false.

Example

```
1   7      "7"
2   0      false
3   0      ''
```

Console Output

```
false
false
false
```

For this reason, the `===` operator is often said to measure **strict equality**.

Just as equality operator `==` has the inequality operator `!=`, there is also a strict inequality operator, `!==`. The boolean expression `a !== b` returns `true` when the two operands are of different types, or if they are of the same type and have different values.

Tip

USE === AND !== WHENEVER POSSIBLE. In this book we will use these strict operators over the loose operators from now on.

5.2.3 Check Your Understanding

Question

What is the result of the following boolean expression?

4 "4"

-
1. true
 2. false
 3. "true"
 4. "false"
-

Question

What is the difference between == and ===?

1. There is no difference. They work exactly the same.
 2. Only === throws an error if its arguments are of different types.
 3. == converts values of different types to be the same type, while === does not.
 4. == works with all data types, while === does not.
-

5.3 Logical Operators

Recall that an operator is one or more characters that carries out an action on its operand(s). In *Data and Variables* we learned about three types of operators:

- Arithmetic operators, such as +, -, *, /, and %.
- The string operator +.
- Compound assignment operators, such as += and -=.

Arithmetic and string operators take number and string operands, respectively, returning values of the same type. Compound assignment operators work similarly with numbers or strings while also reassigning the value of the first, variable operand.

5.3.1 Boolean Operators

In addition to these operators, we learned about comparison operators like ===, <, and others. These operators are part of a larger class known as **boolean operators**, so-called because they return a boolean value (true or false).

Three additional boolean operators allow us to create more complex expressions. These are described below.

Logical AND

A **compound boolean expression** is a boolean expression built out of smaller boolean expressions. JavaScript allows us to create a compound boolean expression using the logical AND operator, `&&`.

The operator takes two operands, and the resulting expression is `true` if *both* operands are `true` individually. If either operand is `false`, the overall expression is `false`.

Example

In English, the `&&` operator mirrors the use of the word “and” (hence the name “logical AND”). A sentence like “Roses are red and violets are blue,” is true as a whole precisely because roses are actually red, and violets are actually blue.

On the other hand, the sentence “Roses are red and violets are green,” is false as a whole. While roses are indeed red, violets are not green.

Let’s see how this works in code.

Example

```
1   7  5  5  3
2   7  5  2  3
3   2  3  'dog'  'cat'
```

Console Output

```
true
false
false
```

In line 1, `7 > 5 && 5 > 3` evaluates to `true` because both `7 > 5` and `5 > 3` are `true` individually.

The expression `7 > 5 && 2 > 3` evaluates to `false` because one of the two expressions, `2 > 3`, is `false`.

Like line 2, line 3 returns `false` because both sub-expressions are `false`. Notice that we can mix and match data types however we like, as long as both sides of the `&&` expression are themselves boolean expressions.

Logical OR

JavaScript’s logical OR operator, `||`, also creates compound boolean expressions. This operator takes two operands, and the resulting expression is `true` if *either* of the operands are `true` individually. If both operands are `false`, the overall expression is `false`.

Example

As with logical AND, logical OR mirrors our experience of English language truth values. The sentence “Pigs can fly or dogs can run,” is true as a whole. Joining the two clauses by “or” requires that only one of them is true in order for the full sentence to be true.

When both of the clauses joined by “or” are `false`, the statement as a whole is `false`. For example, “Pigs can fly or dogs can speak Spanish,” is a `false` statement.

Let's look at some examples in JavaScript.

```
1      7  5   5  3
2      7  5   2  3
3      2  3   'dog'    'cat'
```

Console Output

```
true
true
false
```

Lines 1 and 2 both return `true` because at least one of the comparison expressions joined by `||` is `true`. Line 3 returns `false` because both sub-expressions are `false`.

Warning

The single symbols `&` and `|` are themselves valid JavaScript operators, so accidentally leaving off one symbol when typing `&&` or `||` will not result in an error message.

The operators `&` and `|` are [bitwise operators](#), which are beyond the scope of this course.

Most programmers rarely use `&` and `|`, and it is not important for you to understand them at this point. However, you should *never* use them in place of `&&` and `||`.

Logical NOT

The logical NOT operator, `!`, takes only a single operand and reverses its boolean value.

Example

```
1      true
2      false
```

Console Output

```
false
true
```

The operator `!` (sometimes called “bang”) has the same semantic role as the word “not” in English.

Example

```
1      5  7
2      'dog'    'cat'
```

Console Output

```
true
true
```

5.3.2 Operator Precedence

We now have a number of operators in our toolkit. It is important to understand how these operators relate to each other with respect to **operator precedence**. Operator precedence is the set of rules that dictate in which order the operators are applied.

JavaScript will always apply the logical NOT operator, `!`, first. Next, it applies the arithmetic operators, followed by the comparison operators. The logical AND and OR are applied last.

This means that the expression `x * 5 >= 10 && y - 6 <= 20` will be evaluated so as to first perform the arithmetic and then check the relationships. The `&&` evaluation will be done last. The order of evaluation is the same as if we were to use parentheses to group, as follows:

5	10	6	20
---	----	---	----

While parentheses are not always necessary due to default operator precedence, they make expressions much more readable. As a best practice, we encourage you to use them, especially for more complicated expressions.

The following table lists operators in order of precedence, from highest (applied first) to lowest (applied last). A complete table for the entire language can be found in the [MDN JavaScript Documentation](#).

Table 2: Operator Precedence

Precedence	Category	Operators
(highest)	Logical NOT	<code>!</code>
	Exponentiation	<code>**</code>
	Multiplication and division	<code>*, /, %</code>
	Addition and subtraction	<code>+, -</code>
	Comparison	<code><=, >=, >, <</code>
	Equality	<code>==, !=, ===, !==</code>
	Logical AND	<code>&&</code>
(lowest)	Logical OR	<code> </code>

5.3.3 Truth Tables

Truth tables help us understand how logical operators work by calculating all of the possible return values of a boolean expression. Let's look at the truth table for `&&`, which assumes we have two boolean expressions, A and B, joined by `&&`.

Example

Table 3: Truth Table for `&&`

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Consider the first row of the truth table. This row states that if A is true and B is true, then A `&&` B is true. This is a fact, regardless of what boolean expressions A and B might actually be. The two middle rows demonstrate that if either A or B is false, then A `&&` B is false. (If this idea is hard to grasp, try substituting actual expressions for A and B.)

5.3.4 Check Your Understanding

Question

Complete the table below.

Table 4: Truth Table for ||

A	B	A B
true	true	
true	false	
false	true	
false	false	

Question

Which of the following properly expresses the order of operations (using parentheses) in the following expression?

5 3 10 4 6 11

1. ((5*3) > 10) && ((4+6) === 11)
 2. (5*(3 > 10)) && (4 + (6 === 11))
 3. (((((5*3) > 10) && 4)+6) === 11
 4. ((5*3) > (10 && (4+6))) === 11
-

Question

What is returned by the following boolean expression?

4 3 2 3

1. true
 2. false
 3. "true"
 4. "false"
-

5.4 Conditionals

At the beginning of this chapter, we decided that we wanted to be able to write code that only executes when a given condition is `true`.

Again, here is our motivating example:

Example

Consider a banking application that can remind you when a bill is due. The application will notify you that a bill is due soon, but *only if* the bill has not already been paid.

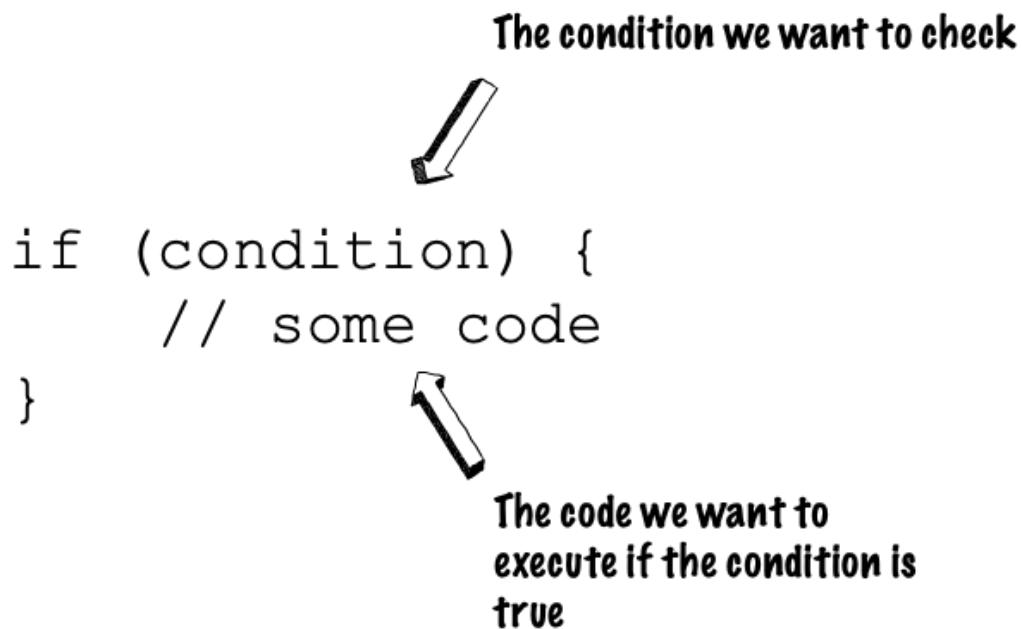
We summarized the condition as follows: Send a notification of an upcoming bill if the statement “the bill is unpaid” is true.

In such a program, JavaScript uses booleans to represent the conditional “the bill is unpaid”. Based on the truth of this statement, the program executes or skips the code for notifying the user.

The JavaScript construct that enables such behavior is a **conditional**.

5.4.1 **if** Statements

The most basic form of a conditional is an **if statement**. Here’s how to create one in JavaScript:



Let’s look at each component of this new syntax.

- The **if** statement consists of a header line and a body. The header line begins with the keyword **if** followed by a boolean expression enclosed in parentheses.
- **condition** is a boolean expression (an expression that evaluates to either `true` or `false`).
- The statements that follow the condition, within `{ }`, make up a **code block**. The code within the brackets `{ }` will be executed if the condition evaluates to true. If the condition evaluates to false, the code within the brackets is ignored.

Here is an explicit example that mimics our banking program.

Example

```
1 let          false
2
3 if
4     "Your bill is due soon!"
5
```

Console Output

```
Your bill is due soon!
```

The message prints because `billHasBeenPaid` is `false`, so `!billHasBeenPaid` evaluates to `true`. If we were to change the value of `billHasBeenPaid` to be `true`, then `!billHasBeenPaid` would evaluate to `false` and the code block would *not* execute.

The condition in an `if` statement can be any boolean expression, such as `name === 'Jack'` or `points > 10` (here, `name` and `points` are variables). Additionally, the code block associated with a conditional can be of any size. This conditional has a code block with two lines of code:

Example

```
1 if      2      0      3
2     "is even"
3     "is greater than 3"
4
```

While not required, the code within a conditional code block is typically indented to make it more readable. Similarly, it is a common convention to place the opening `{` at the end of the first line, and the closing `}` on a line of its own following the last line of the code block.

You should follow such conventions, even though ignoring them will not create an error. To see why, compare the readability of this example, which is functionally equivalent to the one above.

```
1 if      2      0      3
2     "is even"
3     "is greater than 3"
```

Aside from being more aesthetically pleasing, the first version also makes it easier to visually identify the pair of matching curly brackets, which helps prevent syntax errors.

Warning

If the the code block associated with a conditional consists of only one line, then the enclosing curly brackets can be omitted.

However, this is NOT a best-practice, as it makes the logic harder to follow.

```
1 if
2     "Your bill is due soon!"
```

We will use curly brackets for ALL conditional code blocks, and encourage you to do so as well, at least until you become comfortable with reading and writing more complex JavaScript.

5.4.2 else Clauses

An **else clause** can be paired with an **if** statement to specify code that should be executed when the condition is false.

```
if (condition) {  
    // code block #1  
} else {  
    // code block #2  
}
```



**Code that will execute if
the condition is false**

We can use an **else** clause within our bank app to send a message if no bills are currently due.

Example

```
1 let          true  
2 if  
3     "Your bill is due soon!"  
4 else  
5     "Your payments are up to date."  
6  
7
```

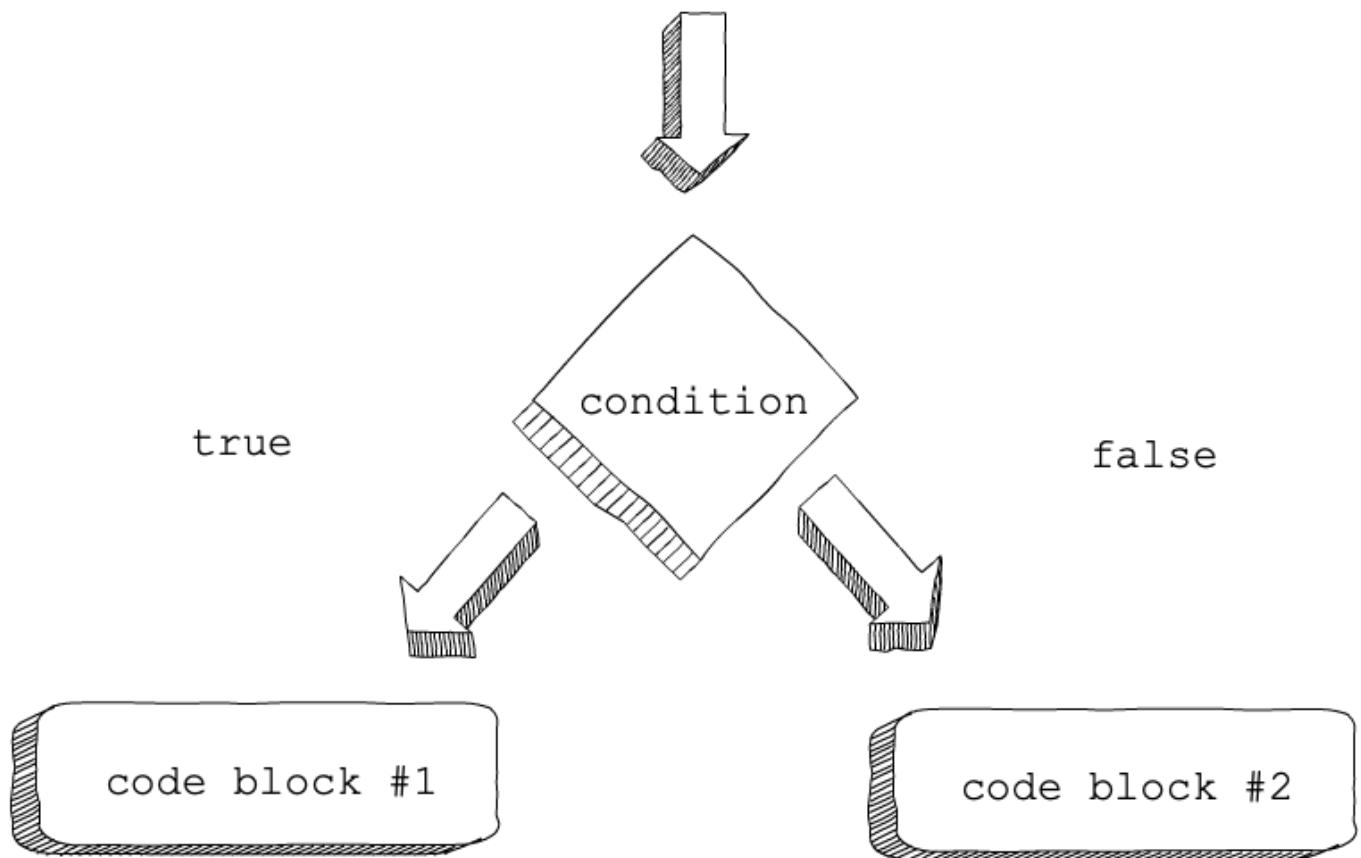
Console Output

```
Your payments are up to date.
```

This structure is known as an **if-else statement**, and it provides a mechanism for **branching**. The flow of the program can take one of two paths when it reaches a conditional, depending on whether the condition is `true` or `false`.

5.4.3 else if Statements

If-else statements allow us to construct two alternative paths. A single condition determines which path will be followed. We can build more complex conditionals using an `else if` clause. These allow us to add additional



conditions and code blocks, which facilitate more complex branching.

Example

```

1 let      10
2 let      20
3
4 if
5         "x is greater than y"
6 else if
7         "x is less than y"
8 else
9         "x and y are equal"
10

```

Console Output

```
x is less than y
```

Let's summarize the flow of execution of this conditional:

1. Line 4 begins the conditional. The boolean expression `x > y` evaluates to `false`, since 10 is not greater than 20. This causes line 5 to be skipped.
2. Line 6 contains an `else if` statement. The boolean expression `x < y` evaluates to `true`, since 10 is less than 20. This triggers the execution of line 7.
3. The code block associated with the `else` clause on lines 8-10 is skipped, because one of the conditions above was true.

As with a simple `if` statement, the `else` clause is optional in this context as well. The following example does not print anything, since both conditions evaluate to false and there is no `else` clause.

```

1 let      10
2 let      10
3
4 if
5         "x is greater than y"
6 else if
7         "x is less than y"
8

```

We can construct conditionals using `if`, `else if`, and `else` with a lot of flexibility. The only rules are:

1. We may not use `else` or `else if` without a preceding `if` statement.
2. `else` and `else if` clauses are optional.
3. Multiple `else if` statements may follow the `if` statement, but they must precede the `else` clause, if one is present.
4. Only one `else` clause may be used.

Regardless of the complexity of a conditional, *no more than one* of the code blocks will be executed.

Example

```
1 let      10
2 let      20
3
4 if
5         "x is greater than y"
6 else if
7         "x is less than y"
8 else if
9         5      0
10        "x is divisible by 5"
11 else if
12         2      0
13         "x is even"
```

Console Output

```
x is less than y
```

Even though both of the conditions `x % 5 === 0` and `x % 2 === 0` evaluate to `true`, neither of the associated code blocks is executed. When a condition is satisfied, the rest of the conditional is skipped.

5.4.4 Check Your Understanding

Question

What does the following code print?

```
1 let      7
2 if      2      1
3         "Launch"
4 else if
5         5
6         "Code"
7 else
8         "LaunchCode"
```

1. "Launch"
 2. "Code"
 3. "Launch"
"Code"
 4. "LaunchCode"
-

5.5 Nested Conditionals

We can write code with more complex branching behavior by combining conditionals and, in particular, by nesting conditionals. Let's see how this works by tackling the following problem.

Example

Write code that prints different messages based on the value of a number variable. If the number is odd, print nothing. If it is even, print “EVEN”. If it is also positive print “POSITIVE”.

Our first attempt at a solution might look like this:

```
1 let      7
2
3 if      2      0
4         "EVEN"
5
6
7 if      0
8         "POSITIVE"
9
```

Console Output

```
POSITIVE
```

We find that the output is POSITIVE, even though 7 is odd and so nothing should be printed. This code doesn’t work as desired because we only want to test for positivity when we already know that the number is even. We can enable this behavior by putting the second conditional *inside* the first.

```
1 let      7
2
3 if      2      0
4         "EVEN"
5
6   if      0
7         "POSITIVE"
8
9
```

repl.it

Try It!

Run the previous example with several different values for num (even, odd, positive, negative) to ensure it works as desired. Nice, huh?

Notice that when we put one conditional inside another, the body of the nested conditional is indented by two tabs rather than one. This convention provides an easy, visual way to determine which code is part of which conditional.

5.5.1 Check Your Understanding

Question

What is printed when the following code runs?

```
1 let      7
2
3 if      2      0
4   if      2      1
5         "odd"
```

(continues on next page)

(continued from previous page)

6
7

1. The code won't run due to invalid syntax
 2. odd
 3. even
 4. The code runs but doesn't print anything
-

Question

Considering the same conditional used in the previous question, which values of num would result in "odd" being printed?

```
1 if      2    0
2   if      2    1
3       "odd"
4
5
```

1. Even values of num.
 2. Odd values of num.
 3. No values. It is impossible for the call to `console.log` to ever run, given the two conditions.
 4. num is 0.
-

5.6 Exercises: Booleans and Conditionals

Attempt these exercises to test your understanding. Don't worry if you struggle while working on them. Struggling and then reviewing the material will help you remember it.

In class, be sure to ask about the topics you do not understand. You are NOT the only person who needs help.

Code exercises 1 & 2 here.

1. Declare and initialize the following variables for our space shuttle:

Variable	Value
<code>engineIndicatorLight</code>	"red blinking"
<code>spaceSuitsOn</code>	true
<code>shuttleCabinReady</code>	true
<code>crewStatus</code>	<code>spaceSuitsOn && shuttleCabinReady</code>
<code>computerStatusCode</code>	200
<code>shuttleSpeed</code>	15000

2. Examine the code below. What will be printed to the console?

Use the value of `engineIndicatorLight` defined above to answer this question.

```

1 if                      "green"
2           "engines have started"
3 else if                  "green blinking"
4           "engines are preparing to start"
5 else
6           "engines are off"
7

```

3. Write conditional expressions to satisfy the safety rules below, using the variables defined from the table above. [Code exercises 3 & 4 here.](#)

1. crewStatus
 - If the value is true, print "Crew Ready"
 - Else print "Crew Not Ready"
2. computerStatusCode
 - If the value is 200, print "Please stand by. Computer is rebooting."
 - Else if the value is 400, print "Success! Computer online."
 - Else print "ALERT: Computer offline!"
3. shuttleSpeed
 - If the value is > 17,500, print "ALERT: Escape velocity reached!"
 - Else if the value is < 8000, print "ALERT: Cannot maintain orbit!"
 - Else print "Stable speed"

4. PREDICT:

Do these code blocks produce the same result? Answer Yes or No.

```

1 if                      200
2           "all systems go"
3 else
4           "WARNING. Not ready"
5

```

```

1 if                      200
2           "WARNING. Not ready"
3 else
4           "all systems go"
5

```

5. The remaining exercises implement conditional code to monitor the shuttle's fuel status. [Code exercises 5 - 7 here.](#)

First, declare and initialize the following variables:

Variable	Value
fuelLevel	21000
engineTemperature	1200

6. Next, implement the checks below using if/else if/else statements.

1. If fuelLevel is above 20000 AND engineTemperature is at or below 2500, print "Full tank. Engines good."

2. If fuelLevel is above 10000 AND engineTemperature is at or below 2500, print "Fuel level above 50%. Engines good."
3. If fuelLevel is above 5000 AND engineTemperature is at or below 2500, print "Fuel level above 25%. Engines good."
4. If fuelLevel is at or below 5000 OR engineTemperature is above 2500, print "Check fuel level. Engines running hot."
5. If fuelLevel is below 1000 OR engineTemperature is above 3500 OR engineIndicatorLight is red blinking print "ENGINE FAILURE IMMINENT!"
6. Otherwise, print "Fuel and engine status pending..."

Try It

Run your code several times to make sure it prints the correct phrase for each set of conditions.

fu-elLevel	engineTempera-ture	engineIndica-torLight	Result
Any	Any	red blinking	ENGINE FAILURE IMMINENT!
21000	1200	NOT red blinking	Full tank. Engines good.
900	Any	Any	ENGINE FAILURE IMMINENT!
5000	1200	NOT red blinking	Check fuel level. Engines running hot.
12000	2600	NOT red blinking	Check fuel level. Engines running hot.
18000	2500	NOT red blinking	Fuel level above 50%. Engines good.

-
7. Final bit of fun!

The shuttle should only launch if the fuel tank is full and the engine check is OK. *However*, let's establish an override command to ignore any warnings and send the shuttle into space anyway!

1. Create the variable `commandOverride`, and set it to be `true` or `false`.
If `commandOverride` is `false`, then the shuttle should only launch if the fuel and engine check are OK.
If `commandOverride` is `true`, then the shuttle will launch regardless of the fuel and engine status.
2. Code the following `if / else` check:
If `fuelLevel` is above 20000 AND `engineIndicatorLight` is NOT red blinking OR `commandOverride` is `true` print "Cleared to launch!"
Else print "Launch scrubbed!"

5.7 Studio: Goal Setting and Getting into the Right Mindset

During this studio, we will ask you to think about your mindset when it comes to facing challenges. Do challenges deter you or delight you? If challenges are something you delight in, you might have a growth mindset.

A **growth mindset** can be powered by one word, “yet”. The word “yet” allows us to acknowledge that we don’t know something AND that we are capable of learning it. With “yet”, the sentence “I don’t understand booleans” becomes “I don’t understand booleans *yet*.”

Growth must occur over an entire career. As technology evolves, even the most senior developers have to learn new skills. Creating a framework for learning, which includes setting achievable goals, is important for being a developer. Technology will change and adapt, and we should learn to do the same!

When setting goals, those goals should be SMART. SMART stands for:

Specific

Measurable

Attainable

Relevant

Time Bound

We will be discussing your goals for this class and your career. While you don’t have to do any prior goal setting or preparation to make the most of this activity, please take the time to write down some goals before class if it would make you more comfortable.

Note

If you have a place where you like to write down your goals and inspiration, such as a journal, feel free to bring it to class!

5.7.1 Activity

After your TA reviews the relevant topics, we will have a discussion covering the following:

1. **Your goals for taking this class.** You probably have many reasons why this class is something you want to complete. Please share 1 to 2 of your goals and keep them SMART!
2. **Your goals for your career.** How does this class assist you in your career and where do you want to go? Please share 1 to 2 goals for your career.
3. **Your inspirational statements.** This can be anything that motivates you from stories to quotes to ideas. Please share 1 to 2.

After the studio, make sure to write your goals and inspirational statements somewhere where you will regularly see them!

5.7.2 Resources

1. *Best Practices: Learning to Code*
2. *The Power of Believing that You can Improve*
3. *What Having a “Growth Mindset” Actually Means*
4. *Golden Rules of Goal Setting*

ERRORS AND DEBUGGING

6.1 What is Debugging?

Programming is a complex process. Since it is done by human beings, errors often occur. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

Some claim that in 1945, a dead moth caused a problem in one of the first computers. The term “bug” has remained in use since, to refer to any issue that prevents a program from working as intended. Wikipedia even has an image of the supposed [first bug](#).

Three kinds of errors can occur in a program: **syntax errors**, **runtime errors**, and **logic errors**. We will first learn about each type of error, and then we will discuss strategies for debugging code and reducing errors.

6.1.1 Beginning Tips for Debugging

Debugging a program requires a different approach compared to writing the original code. As you debug, think of yourself as a detective. Something has gone wrong, and you must use clues, experience, intuition, trial and error, and an inquisitive spirit to solve the problem.

Oftentimes, you will find it tempting to blame errors on JavaScript itself. However, it is far, far more likely that the error is due to an issue with your code. We encourage you to think critically about the code you have written, and whether you may have made an error in writing your code. Even senior developers make basic errors on occasion!

In this chapter, we will discuss in detail common types of bugs, along with some effective strategies for debugging. You will learn to rely on error messages and `console.log` for clues and insight, and over time you will sharpen your debugging skills. We will also discuss how to approach writing code so as to prevent bugs from occurring in the first place.

6.2 Categories of Errors

It is useful to distinguish between categories of errors in order to quickly identify and fix them. Each category manifests itself in a different way, and some strategies may be more useful for certain types of errors.

6.2.1 Stages of JavaScript Execution

In order to understand programming errors it is useful to understand the two stages of code execution.

Parsing

Before code can be run, it must first be parsed, or validated and prepared for execution. This is known as the **parsing stage**, and you can think of it like the pre-flight check for a plane or space craft.

A lot of detailed, low-level tasks are carried out during this process, but it is enough for us to understand that parsing verifies the syntax and structure of the code. .. _error-categories:

Execution

Once our code has been parsed, its syntax has been verified and the program is ready to run. The **execution stage** is when the actions written into our program—printing to the console, prompting the user for input, making calculations, etc.—are actually carried out. You can think of this stage as the plane taking flight.

6.2.2 Syntax Errors

JavaScript can only execute a program if the program is syntactically correct. **Syntax** refers to the structure of a language (spoken, programming, or otherwise) and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with appropriate punctuation.

A **syntax error** is a violation of the formal rules for a given language.

Examples

this sentence contains a syntax error.

So does this one

For most readers of English, a few syntax errors are not a significant problem. Our brains are often flexible enough to determine the intended meaning of a sentence even if it contains one or more syntax errors.

Programming languages are not so forgiving. If there is a single syntax error anywhere in your program, JavaScript will display an error message and quit immediately. Since syntax is validated during the parsing stage, syntax errors are the first we see when running a program.

During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors, and you will find your errors faster.

Try It!

Find the syntax errors in the program.

```
1 let  
2
```

repl.it

Question

What syntax errors did you find? What was the specific error message provided by JavaScript in each case?

6.2.3 Runtime Errors

The second category consists of **runtime errors**, so called because they do not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors occur during the execution phase of a program, so we will only encounter them after the syntax of our program is completely correct.

A common runtime error occurs when we try to use a variable that has not been created yet. This can happen if you misspell the name of a variable, as the following example shows.

Example

```
1 let      "Jack"  
2
```

Console Output

```
ReferenceError: firstname is not defined  
  at evalmachine.<anonymous>:2:13  
  at Script.runInContext (vm.js:107:20)  
  at Object.runInContext (vm.js:285:6)  
  at evaluate (/run_dir/repl.js:133:14)  
  at ReadStream.<anonymous> (/run_dir/repl.js:116:5)  
  at ReadStream.emit (events.js:189:13)  
  at addChunk (_stream_readable.js:284:12)  
  at readableAddChunk (_stream_readable.js:265:11)  
  at ReadStream.Readable.push (_stream_readable.js:220:10)  
  at lazyFs.read (internal/fsstreams.js:181:12)
```

The syntax of our program is correct, but when the program executes, an error occurs at line 2. We attempt to print the value of the variable `firstname`, but such a variable does not exist.

6.2.4 Logic Errors

The third type of error is the **logic error**. If there is a logic error in your program, it will run successfully and not generate any error messages. However, the program will not work as intended.

The characteristic of logic errors is that the program you wrote is not the program you wanted. For example, say you want a program to calculate your daily earnings based on your weekly salary. You might try the following:

Example

```
1 let      600  
2  
3 let      7  
4
```

Console Output

```
85.71428571428571
```

The result surprises you because you thought you were making at least \$100 per day (you work Monday through Friday). According to this program, though, you are making about \$85 per day. The error is a logic one because you

divided your weekly pay by 7. It would have been more accurate to divide your weekly pay by 5, since that is how many days a week you come to work.

Identifying logic errors can be tricky because unlike syntax and runtime problems, there are no error messages to help us identify the issue. We must examine the output of the program and work backward to figure out what it is doing wrong.

6.2.5 Check Your Understanding

Question

Label each of the following as either a syntax, runtime, or logic error.

1. Trying to use a variable that has not been defined.
 2. Leaving off a close parenthesis,), when calling `console.log`.
 3. Forgetting to divide by 100 when printing a percentage amount.
-

6.3 Diagnosing Error Messages

Syntax and runtime errors *always* produce error messages. Reading and understanding error messages is a crucial first step in fixing these types of bugs.

Error messages are your friends. This idea can seem foreign to new programmers, because an error message is a signal that your program is broken. When we are working with a broken program, we might feel frustrated, like we do not fully understand the concepts at hand.

However, the reality is that *all* programmers, no matter how experienced, regularly make simple mistakes. If you run your program and it produces an error message, your first reaction should be, “Great! My program has an error, but I have a helpful message to help me fix it.”

Let’s consider a small program with a couple of syntax errors.

Example

```
let name = Julie;
console.log("Hello, name");
```

While you can spot one or more errors just by looking at the code, let’s examine the error messages produced.

6.3.1 A Syntax Error

Running the program at this stage results in the message:

```
/Users/chris/dev/sandbox/js/syntax.js:2
console.log("Hello, name");
^~~~~~
SyntaxError: Invalid or unexpected token
    at new Script (vm.js:85:7)
```

(continues on next page)

(continued from previous page)

```
at createScript (vm.js:266:10)
at Object.runInThisContext (vm.js:314:10)
at Module._compile (internal/modules/cjs/loader.js:698:28)
at Object.Module._extensions..js (internal/modules/cjs/loader.js:749:10)
at Module.load (internal/modules/cjs/loader.js:630:32)
at tryModuleLoad (internal/modules/cjs/loader.js:570:12)
at Function.Module._load (internal/modules/cjs/loader.js:562:3)
at Function.Module.runMain (internal/modules/cjs/loader.js:801:12)
at internal/main/run_main_module.js:21:11
```

While there is a lot of text in this message, the first few lines tell us everything we need to know.

The first portion identifies where in our code the error exists:

```
console.log("Hello, name);
          ^^^^^^^^^^^^^^
```

For many simple syntax errors, we will quickly be able to spot the mistake once JavaScript points out its location to us.

If knowing the location of the error isn't enough, then next line provides more information:

```
SyntaxError: Invalid or unexpected token
```

This line identifies that actual issue that JavaScript found. It makes it clear that we are dealing with a `SyntaxError`, and it provides a message that describes the issue.

If you are scratching your head at the message, “Invalid or unexpected token,” don’t worry. Programming languages often report errors in ways that are not always easy to decipher at first glance. However, a second look at the line in question helps us make sense of this message.

```
console.log("Hello, name);
          ^^^^^^^^^^^^^^
```

JavaScript is telling us that in the area of `"Hello, name);` it encountered an invalid token. **Token** is a fancy word that means a symbol, variable, or other atomic element of a program. In this case, the invalid token is `"Hello, name);`. JavaScript sees the double-quote character and expects a string. However, the string does not have a closing `",` making it invalid.

Fixing this error gives us a program with correct syntax:

```
1 let
2   "Hello"
```

Note

Error messages may differ depending on where you run your code. The same program run in a [repl.it](#) and Node.js on your computer will generate slightly different error messages. However, these differences are minor and generally unimportant. The main cause of the error will be reported in the same way.

6.3.2 Syntax Errors and Code Highlighting

Most code editors provide a feature known as **syntax highlighting**. Such editors highlight different types of tokens in different ways. For example, strings may be red, while variables may be green. This useful feature gives you a quick, visual way to identify syntax errors.

For example, here is a screenshot of our flawed code taken within an editor at repl.it.

```
1  let name = Julie;
2  console.log("Hello, name");
```

Fig. 1: Screenshot of a program with two syntax errors

Notice that the string Hello is colored red, while *most* of the symbols (=, ;, ., and ()) are colored black. At the end of line 2, however, the final) and ; are both red rather than black. Since we haven't closed the string, the editor assumes that these two symbols are *part of* the string. Since we expect) ; to be black in this editor, the difference in color is a clue that something is wrong with our syntax.

6.3.3 A Runtime Error

Having fixed the syntax error, we can now run our program again. Doing so displays yet another error.

```
Hello
/Users/chris/dev/sandbox/js/syntax.js:1
let name = Julie;
^

ReferenceError: Julie is not defined
  at Object.<anonymous> (/Users/chris/dev/sandbox/js/syntax.js:1:74)
  at Module._compile (internal/modules/cjs/loader.js:738:30)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:749:10)
  at Module.load (internal/modules/cjs/loader.js:630:32)
  at tryModuleLoad (internal/modules/cjs/loader.js:570:12)
  at Function.Module._load (internal/modules/cjs/loader.js:562:3)
  at Function.Module.runMain (internal/modules/cjs/loader.js:801:12)
  at internal/main/run_main_module.js:21:11
```

We have a new error message, this time involving line 1 of our code. We didn't see this error before because it is a runtime error. Due to the syntax error on line 2, the program stopped during the parsing phase. Even though the current error involves the line *before* the original syntax error, the syntax error still gets reported first.

Once again, we are told where the error occurs:

```
let name = Julie;
^
```

There appears to be an issue with the assignment statement. You might be able to see what it is, but let's inspect the error message anyway. Doing so will help us understand JavaScript errors more generally.

The message is:

```
ReferenceError: Julie is not defined
```

The type of error is `ReferenceError`. If we search the web for "JS ReferenceError" then one of the first results is the [MDN documentation for ReferenceError](#). No need to read the entire document, however. The first sentence on this page tells us what we need to know:

The `ReferenceError` object represents an error when a non-existent variable is referenced.

This information, along with the rest of the message, "Julie is not defined," makes it clear what JavaScript is complaining about. The error message is saying, *Hey, check your variables!*

To us, we see that we forgot to enclose the string `Julie` in quotes, because we know that we intended to assign the variable name a string value. However, to JavaScript there is nothing in the program to indicate that `Julie` should be a string. In fact, JavaScript sees `Julie` as a variable. Since there is no such defined variable in our program, it returns a `ReferenceError`.

This is one of many examples when we, as humans, describe the same error slightly differently than JavaScript. Usually, neither description is better than the other. Humans and computers simply view information differently.

6.4 Error Types

An **error type** is the classification that JavaScript uses to group errors based on their cause. In future lessons, we will learn that an error type is actually something called a **built-in object**. For now, understanding the different types of errors will help us become faster at debugging.

Each error that JavaScript reports has an error type, and the type is included in the error message. For example, *an earlier message* reported the error type as `SyntaxError`.

```
/Users/chris/dev/sandbox/js/syntax.js:2
console.log("Hello, name);
^~~~~~
```

SyntaxError: Invalid or unexpected token
at new Script (vm.js:85:7)
at createScript (vm.js:266:10)
at Object.runInThisContext (vm.js:314:10)
at Module._compile (internal/modules/cjs/loader.js:698:28)
at Object.Module._extensions..js (internal/modules/cjs/loader.js:749:10)
at Module.load (internal/modules/cjs/loader.js:630:32)
at tryModuleLoad (internal/modules/cjs/loader.js:570:12)
at Function.Module._load (internal/modules/cjs/loader.js:562:3)
at Function.Module.runMain (internal/modules/cjs/loader.js:801:12)
at internal/main/run_main_module.js:21:11

We have now seen two error types, `ReferenceError` and `SyntaxError`. There are several other error types in JavaScript, such as `TypeError` and `RangeError`.

The following table describes all JavaScript error types. Some of these relate to coding concepts we have not covered yet, but we include them here as a reference for future use.

Table 1: JavaScript Error Types

Error Type	Description	Example of code triggering the error	Example description
SyntaxError	Occurs when trying to parse syntactically invalid code.	"hello ^"	The call to <code>console.log</code> does not have a required close parenthesis.
ReferenceError	Occurs when a non-existent variable is used/referenced. 2	<code>let</code> ^ "Jack" ^	The variable <code>firstname</code> does not exist; it is a misspelling of <code>firstName</code> .
TypeError	Occurs when trying to use a value in an invalid way.	1	The numeric value 1 is not a function, so trying to use it as one results in <code>TypeError: 1 is not a function</code> .
RangeError	Occurs when passing an invalid value to a function.	<code>let</code> ^1	The constructor function <code>Array(n)</code> creates an empty array of length n. It is not possible to create an array with negative length, so the code results in <code>RangeError: Invalid array length</code> .
URIError	Occurs when improperly using a global URI-handling function. ('URI' = Uniform Resource Identifier)	'%'	The % character is used to encode characters not otherwise allowed in URIs, such as spaces (%20). If an invalid character encoding is given, a <code>URIError</code> results.
Error	The type from which all other errors are built. It can be used to generate programmer-triggered and programmer-defined errors.	<code>throw</code> ^ "Something bad ^ happened!"	Manually triggers an error with the given message.

Each time you encounter a new error type, take the time to understand what it is, and what JavaScript is trying to tell you. Remember, **error messages are your friends!**

6.5 Debugging Logic Errors

We can debug runtime and syntax errors using the error messages produced. Logic errors, however, do not generally produce error messages. This sometimes makes them tougher to debug.

While we can't provide a step-by-step approach that applies to every possible logic error, we *can* give you some solid strategies. Two such strategies—using debugger tools and writing tests—will be covered in future lessons. In this section, we start with a basic and effective way to debug logic errors.

6.5.1 Printing Values

When your code runs but doesn't produce the expected results, it is important to check the values of the variables being used.

Let's look at a program that has a logical bug.

```

1 const          'readline-sync'
2
3 let           'Temp in degrees C:'
4 let           273.15
5
6   'Degrees K:'
```

repl.it

This program asks the user for a temperature in degrees celsius and attempts to convert it to degrees Kelvin. Degrees Kelvin differs from degrees celsius by 273.15. So if we enter 100 (in celsius) we should see a converted value of 373.15 (in Kelvin). However, running the program as-is and entering 100 gives the message:

```
Temp in degrees C: 100
Degrees K: 100273.15
```

This is clearly incorrect. But the program does not generate an error, so it is not immediately clear what the issue is. To figure it out, we'll use `console.log` to see what the values of key variables are when the program runs.

Let's first make sure that the `degreesC` variable looks like it should by adding a `console.log` statement just after we create this variable.

```

1 const          'readline-sync'
2
3 let           'Temp in degrees C:'
4
5 let           273.15
6
7   'Degrees K:'
```

Running this with an input of 100 gives the output:

```
Temp in degrees C: 100
100
Degrees K: 100273.15
```

The second line is the value of `degreesC`, which appears to be correct. But the final answer is still incorrect, so we need to keep digging for more information.

Looking at the line in which we set `degreesK`, we see that we use `degreesC` as a numeric value in our calculation. Let's see what the data type of `degreesC` is. In the end, we want it to be a number.

```

1 const          'readline-sync'
2
3 let           'Temp in degrees C:'
4   typeof
5 let           273.15
6
7   'Degrees K:'
```

Running this with an input of 100 gives the output:

```
Temp in degrees C: 100
string
Degrees K: 100273.15
```

That's it! The variable `degreesC` has the value `100`, but it is a string rather than a number. So when we set `degreesK` with the formula `degreesC + 273.15`, we are actually performing string concatenation instead of addition: `"100" + 273.15` is `"100273.15"`.

We can fix our program by converting the user's input to the number data type.

```
1 const          'readline-sync'
2
3 let           'Temp in degrees C: '
4
5 let           273.15
6
7   'Degrees K:'
```

Running this with an input of `100` gives the output:

```
Temp in degrees C: 100
Degrees K: 373.15
```

Note that after debugging we removed all of our `console.log` statements. Be sure to do the same when using this debugging technique.

6.6 How to Avoid Debugging

While debugging is an unavoidable part of programming, you can reduce the number of bugs you encounter by working carefully.

6.6.1 Start Small

This is probably the best piece of advice for programmers at every level. It can be tempting to sit down and write an entire program all at once. However, this leaves a large number of possibilities when the program does not work. The errors could be hiding anywhere in the code. The more code, the more possibilities exist. Where to start? How to figure out what went wrong?

When you start work on a large program, break the process down into smaller steps. Begin coding one very small part—even if that's just 2 lines of code. Then make sure the program runs properly before adding the next small part.

Regularly running your code is quick and easy, and doing so gives you immediate feedback about how the code will run.

6.6.2 Keep It Working

Once you have a small part of your program working, the next step is to figure out something small to add to it. If you keep adding small pieces to the program, one at a time, it is much easier to figure out what went wrong. Any error that occurs was almost certainly introduced by the last line or two of code that was added. Less new code makes it easier to locate the problem.

Here is your new mantra, “Get something working and keep it working.” Repeat this throughout your career as a programmer. It’s a great way to avoid frustration and reduce stress while creating amazing (and working) code.

Get something working and keep it working.

Research has shown that with every little success, your brain releases a tiny bit of a chemical that makes you happy. So you can keep yourself happy and make programming more enjoyable by creating lots of small victories for yourself.

6.7 Asking Good Questions

If you still cannot find the bug in your code after using the strategies outlined in this chapter, do not hesitate to reach out to other programmers.

Whether you are asking your teaching assistant, another student, or a stranger in an online forum, you should first be able to answer the 3 questions outlined below.

Not only will these questions help others assist you more directly, but they may just lead you to the answer yourself!

6.7.1 What is the problem with your code?

Describe the error you are experiencing with as much detail as possible.

Bad: “My program is broken.”

Bad: “I’m getting this error.”

Good: “There is a ReferenceError on line 23, but it’s not clear to me what’s causing it.”

6.7.2 What have you done to try to address the problem?

Another programmer can glean a lot of information by hearing what you have already tried.

Bad: Asking for help immediately.

Bad: Using trial and error without any specific direction.

Good: “I added user input validation, but am still seeing the problem.”

6.7.3 Where have you looked for an answer?

Bad: “I haven’t looked online at all.”

Bad: “I Googled ‘js range error’ and didn’t see anything.”

Good: “I Googled ‘js range error boolean expression’ and found a question on Stack Overflow that seemed relevant. I tried the recommended solution, but it didn’t fix my problem.”

6.8 Exercises: Debugging

Avast, ye scurvy dogs! We be needn’ ta fix yonder code!

The cap’n in charge of clearing the shuttle for launch(code) be out with an illness, and ye be the next tech in line.

Yer job is to evaluate the code and fix any errors. Remember, the lives of the crew rest squarely upon yer shoulders. Happy second day on the job!

Yer directions are as follows:

1. Launch the shuttle *only if* the fuel, crew and computer all check out OK.
2. If a check fails, print that information to the console and scrub the launch (then scrub the deck).
3. If all checks be successful, print a countdown to the console, then bellow “Liftoff!”

6.8.1 Debugging Practice

1. Fix **syntax errors** first. Run the following code as-is and squint yer eyes at the error message. Fix the mistake, and then re-run the code to check it.

```
1 let          false
2 let          17000
3
4 if          20000
5   'Fuel level cleared.'
6   true
7 else
8   'WARNING: Insufficient fuel!'
9   false
10
```

Fix it at [repl.it](#)

2. The next block of code hides two syntax errors. Run the code as-is to find the mistakes. *Tip:* Don't be too hasty, Matey! Only ONE error will be flagged at a time. Fix that ONE problem, and then re-run the code to check yer work. Avoid trying to fix multiple issues at once.

```
let launchReady = false;
let crewStatus = true;
let computerStatus = 'green';

if (crewStatus && computerStatus === 'green') {
  console.log('Crew & computer cleared.');
  launchReady = true;
} else {
  console.log('WARNING: Crew or computer not ready!');
  launchReady = false;
}

if (launchReady) {
  console.log("10, 9, 8, 7, 6, 5, 4, 3, 2, 1...");
  console.log("Fed parrot...");
  console.log("Ignition...");
  console.log("Liftoff!");
} else {
  console.log("Launch scrubbed.");
}
```

Fix it at [repl.it](#)

3. Fix **runtime errors** next. Remember to examine the error message for clues about what is going wrong. Pay close attention to any line numbers mentioned in the message - these will help ye locate and repair the mistake in the code.

```
1 let          false
2 let          17000
3
```

(continues on next page)

(continued from previous page)

```

4      20000
5      'Fuel level cleared.'
6      true
7
8      'WARNING: Insufficient fuel!'
9      false
10

```

Fix it at repl.it

4. *Arrr!* Now find and fix the runtime error in a longer code sample.

```

1      false
2      27000
3
4      20000
5      'Fuel level cleared.'
6      true
7
8      'WARNING: Insufficient fuel!'
9      false
10
11
12      "10, 9, 8..."
13      "Fed parrot..."
14      "6, 5, 4..."
15      "Ignition..."
16      "3, 2, 1..."
17      "Liftoff!"
18
19      "Launch scrubbed."
20
21

```

Fix it at repl.it

5. Solve **logic errors** last. Logic errors do not generate warning messages or prevent the code from running, but the program still does not work as intended. (Refer to *debugging logic errors* if ye need to review).

1. First, run this sample code as-is and examine the output.

```

1      false
2      17000
3      true
4      'green'
5
6      20000
7      'Fuel level cleared.'
8      true
9
10     'WARNING: Insufficient fuel!'
11     false
12
13
14     'green'
15     'Crew & computer cleared.'
16     true
17

```

(continues on next page)

(continued from previous page)

```
18      'WARNING: Crew or computer not ready!'
19          false
20
21
22 if
23     '10, 9, 8, 7, 6, 5, 4, 3, 2, 1...'
24     'Liftoff!'
25 else
26     'Launch scrubbed.'
```

Run it at [repl.it](#)

Should the shuttle have launched? Did it?

2. Let's break the code down into smaller chunks. Consider the first if/else block below. Add `console.log(launchReady)` after this block, then run the program.

```
1 let      false
2 let      17000
3
4 if      20000
5     'Fuel level cleared.'
6     true
7 else
8     'WARNING: Insufficient fuel!'
9     false
10
```

Run it at [repl.it](#)

Given the `fuelLevel` value, should `launchReady` be `true` or `false` after the check? Is the program behaving as expected?

3. Now consider the second if/else block. Add another `console.log(launchReady)` after this block and run the program.

```
1 let      false
2 let      true
3 let      'green'
4
5 if      'green'
6     'Crew & computer cleared.'
7     true
8 else
9     'WARNING: Crew or computer not ready!'
10    false
11
```

Run it at [repl.it](#)

Given `crewStatus` and `computerStatus`, should `launchReady` be `true` or `false` after this check? Is the program behaving as expected?

4. Now consider both if/else blocks together (keeping the added `console.log` lines). Run the code and examine the output.

```
1 let          false
2 let          17000
3 let          true
4 let          'green'

5
6 if          20000
7   'Fuel level cleared.'
8     true
9 else
10   'WARNING: Insufficient fuel!'
11   false

12
13
14
15 if          'green'
16   'Crew & computer cleared.'
17     true
18 else
19   'WARNING: Crew or computer not ready!'
20   false
21
22
```

Run it at [repl.it](https://repl.it/@joshuawilliams/SpaceShuttleLaunchCheck)

Given the values for `fuelLevel`, `crewStatus` and `computerStatus`, should `launchReady` be `true` or `false`? Is the program behaving as expected?

5. Ahoy, Houston! We spied a problem! The value of `launchReady` assigned in the first `if/else` block got changed in the second `if/else` block. Dangerous waters, Matey. Since the issue is with `launchReady`, ONE way to fix the logic error is to use a different variable to store the fuel check result. Update yer code to do this. Verify that yer change works by updating the `console.log` statements.

Fix it at [repl.it](https://repl.it/@joshuawilliams/SpaceShuttleLaunchCheck#fix)

6. Almost done, so wipe the sweat off yer brow! Add a final `if/else` block to print a countdown and “Liftoff!” if all the checks pass, or print “Launch scrubbed” if any check fails.

Blimey! That's some good work. Now go feed yer parrot.

STRINGING CHARACTERS TOGETHER

7.1 Strings as Collections

Throughout the first chapters of this book we have used strings to represent words or phrases we wanted to print out. Our definition of a string was simple: a string is a sequence of characters inside quotes.

In this chapter we explore strings in much more detail. Strings come with a special group of operations that can be carried out on them, known as methods. Strings are also what is called a collection data type. Let's look at what this means.

7.1.1 Collection Data Types

Data types that are comprised of smaller pieces are called **collection data types**, or simply **collection types**. Depending on what we are doing, we may want to treat a value of a collection data type as a single entity (the whole collection), or we may want to access its parts.

A **character** is a string that contains exactly one element, such as '`a`', '`?`', or even '' (a single space character).

Note

Some programming languages, such as Java and C, represent characters using their own data type. For example, Java has the data type `char`. JavaScript, however, does not consider strings and characters to be different types.

We can think of strings as being built out of characters. In this way, strings can be broken down into smaller pieces.

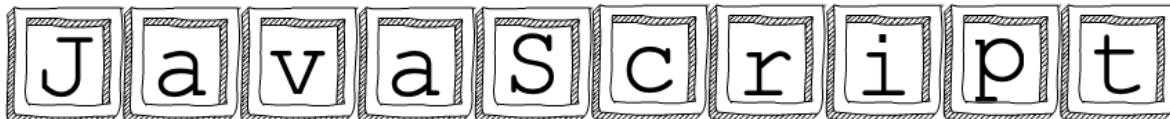


Fig. 1: A string is made up of characters, which are strings of length 1.

Strings are made up of smaller pieces, characters. Other data types, like `number` and `boolean`, are not composed of any smaller parts.

7.1.2 Ordered Collections

We defined strings as *sequential* collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right. The string "LaunchCode" is different from the string "CodeLaunch", even though they contain the exact same characters.

Collection types that allow their elements to be ordered are known as **ordered collections**, for reasons that will become clear to you very soon.

7.2 Bracket Notation

Understanding strings as sequential collections of characters gives us much more than just a mental model of how they are structured. JavaScript provides a rich collection of tools—including special syntax and operations—that allows us to work with strings.

Bracket notation is the special syntax that allows us to access the individual characters that make up a string. To access a character, we use the syntax `someString[i]`, where `i` is the **index** of the character we want to access. String indices are integers representing the position of a character within a given string, and they start at 0. Thus, the first character of a string has index 0, the second has index 1, and so on.

Consider the string "JavaScript". The "J" has index 0, the first "a" has index 1, "v" has index 2, and so on.



Fig. 2: The indices of a string.

An expression of the form `someString[i]` gives the character at index `i`.

Example

This program prints out the initials of the person's name.

```
1      let      "Brendan Eich"
2      let      0
3      let      8
4
5      let      "JavaScript was created by somebody with initials "
6          "."
7          "."
8
9
```

Console Output

```
JavaScript was created by somebody with initials B.E.
```

What happens if we try to access an index that doesn't exist, for example -1 or an index larger than the length of the string?

Try It!

```
1 let          "Brendan Eich"  
2  
3           1  
4           42
```

repl.it

Question

What does an expression using bracket notation evaluate to when the index is invalid (the index does not correspond to a character in the string)?

7.2.1 Check Your Understanding

Question

If phrase = 'Code for fun', then phrase[2] evaluates to:

1. "o"
 2. "d"
 3. "for"
 4. "fun"
-

Question

Which of the following returns true given myStr = 'Index'? Choose all correct answers.

1. myStr[2] === 'n';
 2. myStr[4] === 'x';
 3. myStr[6] === ' ';
 4. myStr[0] === 'I';
-

Question

What is printed by the following code?

```
1 let      "JavaScript rocks!"  
2
```

8

1. "p"
 2. "i"
 3. "r"
 4. "t"
-

7.3 Strings as Objects

Beyond bracket notation, there are many other tools we can use to work with strings. Talking about these tools requires some new terminology.

7.3.1 Object Terminology

In JavaScript, strings are objects, so to understand how we can use them in our programs, we must first understand some basics about objects.

An **object** is a collection of related data and operations. An operation that can be carried out on an object is known as a **method**. A piece of data associated with an object is known as a **property**.

Example

Suppose we had a `square` object in JavaScript. (While no such object is built into JavaScript, we will learn how we could make one in a later chapter.)

Since a square has four sides of the same length, it should have a property to represent this length. This property could be called `length`. For a given square, it will have a specific value, such as 4.

Since a square has an area, it should have a method to calculate the area. This method could be called `area`, and it should calculate the area of a square using its `length` property.

You can think of methods and properties as functions and variables, respectively, that “belong to” an object. Properties and methods are accessed using **dot notation**, which dictates that we use the object name, followed by a `.`, followed by the property or method name. When using a method, we must also use parentheses as we do when calling regular functions.

Example

Returning to the `square` example, we can access its length by typing `square.length`.

We can calculate the area by calling `square.area()`.

Referencing `length` or `area` by itself in code *does not* give you the value of `square.length` or carry out the calculation in `square.area()`. It does not make sense to refer to a property or method without also referring to the associated object. Typing simply `length` or `area()` is ambiguous. There might be multiple squares, and it would be unclear which one you were asking about.

Example

We have already encountered one object, the built-in object `console`, which we use to output messages.

```
typeof
```

Console Output

```
object
```

JavaScript reports that the type of `console` is indeed `object`.

When calling `console.log`, we are calling the `log` method of the `console` object.

We will learn quite a bit more about objects in this course, including how to use objects to create your own custom data types. This powerful JavaScript feature allows us to bundle up data and functionality in useful, modular ways.

7.3.2 Strings Are Objects

The fact that strings are objects means that they have associated data and operations, or properties and methods as we will call them from now on.

Every string that we work with will have the same properties and methods. The most useful string property is named `length`, and it tells us how many characters are in a string.

Example

```
1 let      "Grace"
2 let      "Hopper"
3
4         "has"           "characters"
5         "has"           "characters"
```

Console Output

```
Grace has 5 characters
Hopper has 6 characters
```

Every string has a `length` property, which is an integer.

The `length` property is the only string property that we will use, but there are many useful string methods. We will explore these in depth in the section *String Methods*, but let's look at one now to give you an idea of what's ahead.

The `toLowerCase()` string method returns the value of its string in all lowercase letters. Since it is a method, we must precede it with a specific string in order to use it.

Example

```
1 let      "LaunchCode"
2
3
4
```

Console Output

```
launchcode
LaunchCode
```

Notice that `toLowerCase()` does not alter the string itself, but instead *returns* the result of converting the string to all lowercase characters. In fact, it is not possible to alter the characters within a string, as we will now see.

7.3.3 Check Your Understanding

Question

Given `word = 'Rutabaga'`, why does `word.length` return the integer 8, but `word[8]` is `undefined`?

7.4 String Immutability

If an object cannot be changed, we say that it is **immutable**. Strings are immutable, which means we cannot change the individual characters within a given string. While we can access individual characters using bracket notation, attempting to change individual characters simply does not work.

Example

```
1 let      "Launchcode"
2
3
4   6      "C"
5
```

Console Output

```
Launchcode
Launchcode
```

We attempted to change the value of the character at index 6 from '`c`' to '`C`', by using an assignment statement along with bracket notation on line 4 (perhaps to align with official LaunchCode branding guidelines). However, this change clearly did not take place. In many programming languages strings are immutable, and while trying to change a string in some languages results in an error, JavaScript simply ignores our request to alter a string.

It is important to notice that immutability applies to string *values* and not string variables.

Example

We can set a variable containing a string to a different value.

```
1 let      "Launchcode"
2           "LaunchCode"
3
4
```

Console Output

```
LaunchCode
```

In this example, the change made on line 2 is carried out. The difference between this example and the one above is that here we are modifying the value that the variable is storing, and not the string itself. Using our visual analogy of a variable as a label that “points at” a value, the second example has the following effect:

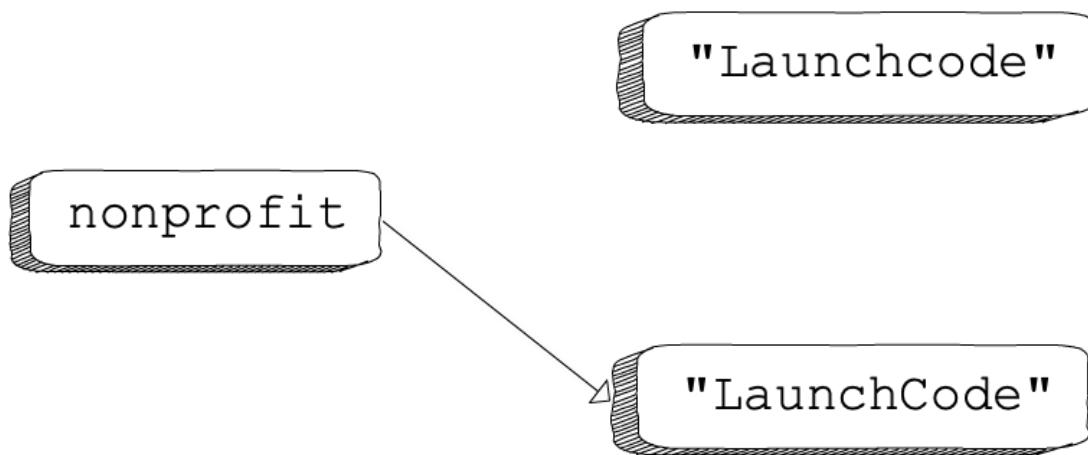


Fig. 3: When the value of a variable storing a string is changed, the variable then points to a new value, with the old value remaining unchanged.

7.4.1 Check Your Understanding

Question

Given `pet = 'cat'`, why do the statements `console.log(pet + 's');` and `pet += 's';` NOT violate the immutability of strings?

7.5 String Methods

JavaScript provides many useful methods for string objects. Recall that a method is a function that “belongs to” a specific object. Methods will typically result in some operation being carried out on the data within an object. For strings, this means that our methods will typically transform the characters of the given string in some way.

As we have learned, strings are immutable. Therefore, string methods will not change the value of a string itself, but instead will *return* a new string that is the result of the given operation.

We saw this behavior in the `toLowerCase` example.

Example

```
1 let      "LaunchCode"
2
3
4
```

Console Output

```
launchcode
LaunchCode
```

While `nonprofit.toLowerCase()` evaluated to `"launchcode"`, the value of `nonprofit` was left unchanged. This will be case for each of the string methods.

7.5.1 Common String Methods

Here we present the most commonly-used string methods. You can find documentation for other string methods at:

- [W3Schools](#)
- [MDN](#)

Table 1: Common String Methods

Method	Syntax	Description
<code>in-</code> <code>dexOf</code>	<code>stringName.</code> <code>indexOf(substr)</code>	Returns the index of the first occurrence of the substring in the string, and returns -1 if the substring is not found.
<code>toLow-</code> <code>erCase</code>	<code>stringName.toLowerCase()</code>	Returns a copy of the given string, with all uppercase letters converted to lowercase.
<code>toUp-</code> <code>per-</code> <code>Case</code>	<code>stringName.toUpperCase()</code>	Returns a copy of the given string, with all lowercase letters converted to uppercase.
<code>trim</code>	<code>stringName.trim()</code>	Returns a copy of the given string with the leading and trailing whitespace removed.
<code>re-</code> <code>place</code>	<code>stringName.</code> <code>replace(searchChar,</code> <code>replacementChar)</code>	Returns a copy of <code>stringName</code> , with the first occurrence of <code>searchChar</code> replaced by <code>replacementChar</code> .
<code>slice</code>	<code>stringName.slice(i, j)</code>	Return the substring consisting of characters from index <code>i</code> through index <code>j-1</code> .

Tip

String methods can be combined in a process called **method chaining**. Given `word = 'JavaScript';`, `word.toUpperCase()` returns `JAVASCRIPT`. What would `word.slice(4).toUpperCase()` return? Try it at [repl.it](#).

7.5.2 Check Your Understanding

Follow the links in the table above for the `replace`, `slice`, and `trim` methods. Review the content and then answer the following questions.

Question

What is printed by the following code?

```
1 let      "JavaScript"
2           'J'  'Q'
3   0 5
```

-
- 1. "JavaScript"
 - 2. "QavaScript"
 - 3. "QavaSc"
 - 4. "Qavas"
-

Question

Given `language = 'JavaScript';`, what does `language.slice(1, 6)` return?

- 1. "avaScr"
 - 2. "JavaSc"
 - 3. "avaSc"
 - 4. "Javas"
-

Question

What is the value of the string printed by the following program?

```
1 let      " The LaunchCode Foundation "
2 let
```

-
- 1. " The LaunchCode Foundation "
 - 2. "The LaunchCode Foundation"
 - 3. "TheLaunchCodeFoundation"
 - 4. " The LaunchCode Foundation"
-

7.6 Encoding Characters

If you had microscope powerful enough to view the data stored on a computer's hard drive, or in its memory, you would see lots of 0s and 1s. Each such 0 and 1 is known as a **bit**. A bit is a unit of measurement, like a meter or a pound. Collections of computer data are measured in bits; every letter, image, and pixel you interact with on a computer is represented by bits.

We work with more complex data when we program, including numbers and strings. This section examines how such data is represented within a computer.

7.6.1 Representing Numbers

A **byte** is a set of 8 bits. Bytes look like 00101101 or 11110011, and they represent a **binary number**, or a base-2 number. A binary number is a number representation that uses only 0s and 1s. The numbers that you are used to, which are built out of the integers 0...9, are **decimal numbers**, or base-10 numbers.

Since each bit can have one of two values, each byte can have one of $2^8 = 256$ different values.

It may not be obvious, but every decimal integer can be represented as a binary integer, and vice versa. There are 256 different values a byte may take, each of which can be used to represent a decimal integer, from 0 to 255.

Note

We will not go into binary to decimal number conversion. If you are interested in learning more, there are [many tutorials online](#) that can show you the way.

In this way, the bits in a computer can be viewed as integers. If you want to represent values greater than 255, just use more bits!

7.6.2 Representing Strings

Strings are collections of characters, so if we can represent each character as a number, then we'll have a way to go from a string to a collection of bits, and back again.

Character Encodings

Unlike the natural translation between binary and decimal numbers, there is no natural translation between integers and characters. For example, you might create a pairing of 0 to a, 1 to b, and so on. But what integer should be paired with \$ or a tab? Since there is no natural way to translate between characters and integers, computer scientists have had to make such translations up. Such translations are called **character encodings**.

There are many different encodings, some of which continue to evolve as our use of data evolves. For instance, the most recent versions of the Unicode character encoding include emoji characters, such as .

The ASCII Encoding

Most of the characters that you are used to using—including letters, numbers, whitespace, punctuation, and symbols—are part of the **ASCII** (pronounced *ask-ee*) character encoding. This standard has changed very little since the 1960s, and it is the foundation of all other commonly-used encodings.

Note

ASCII stands for American Standard Code for Information Interchange, but most programmers never remember that, so you shouldn't try to either.

ASCII provides a standard translation of the most commonly-used characters to one of the integers 0...127, which means each character can be stored in a computer using a single byte.

ASCII maps a to 97, b to 98, and so on for lowercase letters, with z mapping to 122. Uppercase letters map to the values 65 through 90. The other integers between 0 and 127 represent symbols, punctuation, and other assorted odd characters. This scheme is called the **ASCII table**, and rather than replicate it here, we refer you to an [excellent one online](#).

In summary, strings are stored in a computer using the following process:

1. Break a string into its individual characters.
 2. Use a character encoding, such as ASCII, to convert each of the characters to an integer.
 3. Convert each integer to a series of bits using decimal-to-binary integer conversion.
-

Fun Fact

JavaScript uses the UTF-16 encoding, which includes ASCII as a subset. We will rarely need anything outside of its ASCII subset, so we will usually talk about “ASCII codes” in JavaScript.

7.6.3 Character Encodings in JavaScript

JavaScript provides methods to convert any character into its ASCII code and back.

The string method `charCodeAt` takes an index and returns the ASCII code of the character at that index.

Example

```
1 let          "LaunchCode"  
2  
3 for  let      0  
4  
5
```

Console Output

```
76  
97  
117  
110  
99  
104  
67  
111  
100  
101
```

To convert an ASCII code to an actual character, use `String.fromCharCode()`.

Example

```
1 let          76  97  117  110  99  104  67  111  100  101  
2 let          ""  
3  
4 for  let      0  
5  
6  
7  
8
```

Console Output

```
LaunchCode
```

7.7 Special Characters

Aside from letters, numbers, and symbols, there is another class of characters that we will occasionally use in strings, known as **special characters**. These characters consist of special character combinations that all begin with \ backslash). They allow us to include characters in strings that would be difficult or impossible to include otherwise, such as Unicode characters that are not on our keyboards, control characters, and whitespace characters.

The most commonly-used special characters are \n and \t, which are the newline and tab characters, respectively. They work as you would expect.

Example

```
"A message\nbroken across lines,\n\tand indented"
```

Console Output

```
A message
broken across lines,
    and indented
```

We can also represent Unicode characters (most of which aren't on a normal keyboard) using special character combinations of the form \uXXXX, where the Xs are combinations referenced by the [Unicode table](#). This allows us to use character sets other than the basic Latin characters that English is based on, such as Greek, Cyrillic, and Arabic, as well as a wider array of symbols.

Example

```
"The interrobang character, \u203d, combines ? and !"
```

Console Output

```
The interrobang character, ?, combines ? and !
```

We can also use the backslash, \, to include quotes within a string. This is known as **escaping** a character.

Example

```
"\"The dog's favorite toy is a stuffed hedgehog,\" said Chris"
```

Console Output

```
"The dog's favorite toy is a stuffed hedgehog," said Chris
```

7.7.1 Check Your Understanding

Question

Which of the options below print 'Launch' and 'Code' on separate lines?

1. `console.log('Launch\nCode');`
 2. `console.log('Launch/nCode');`
 3. `console.log('Launch', 'Code');`
 4. `console.log('Launch\tCode');`
 5. `console.log('Launch/tCode');`
-

7.8 Template Literals

Earlier, we used *concatenation* to combine strings and variables together in order to create specific output:

Example

```
1 let
2 let      9
3
4     "Next year, "           " will be "      1      "."

```

Console Output

```
Next year, Jack will be 10.
```

Unfortunately, this process quickly gets tedious for any output that depends on multiple variables. Often, concatenation requires multiple test runs of the code in order to check for syntax errors and proper spacing within the output. Fortunately, JavaScript offers us a better way to accomplish this process.

Template literals allow for the automatic insertion of expressions (including variables) into strings.

While normal strings are enclosed in single or double quotes (' or "), template literals are enclosed in back-tick characters, ` . Within a template literal, any expression surrounded by \${ } will be evaluated, with the resulting value included in the string.

Example

Template literals allow for variables and other expressions to be directly included in strings.

```
1 let      "Jack"
2 let      9
3
4     `Next year, ${      } will be ${      1} .`
```

Console Output

```
Next year, Jack will be 10.
```

Besides allowing us to include data in strings in a cleaner, more readable way, template literals also allow us to easily create multi-line strings without using string concatenation or special characters.

Example

```
1 let      `The mind chases happiness.  
2 The heart creates happiness.  
3 The soul is happiness  
4 And it spreads happiness  
5 All--where.  
6  
7 - Sri Chinmoy`  
8  
9
```

Console Output

```
The mind chases happiness.  
The heart creates happiness.  
The soul is happiness  
And it spreads happiness  
All--where.  
  
- Sri Chinmoy
```

Note

The ECMAScript specifications define the standard for JavaScript. The 6th edition, known as ES2015, added template literals. Not only are template literals relatively new to JavaScript, but you may encounter environments—such as older web browsers—where they are not supported.

7.8.1 Check Your Understanding

Question

Mad Libs are games where one player asks the group to supply random words (e.g. “Give me a verb,” or, “I need a color”). The words are substituted into blanks within a story, which is then read for everyone’s amusement. In elementary school classrooms, giggles and hilarity often ensue. TRY IT!

Refactor the following code to replace the awkward string concatenation with template literals. Be sure to add your own choices for the variables.

```
1 let  
2 let  
3 let  
4 let  
5 let  
6  
    "JavaScript provides a "           " collection of tools -- including "   
    " syntax and "                   " -- that allows "       " to "      "  
    "with strings."                  " "
```

repl.it

7.9 Exercises: Strings

1. Identify the result for each of the following statements:

1. 'JavaScript'[8]
 2. "Strings are sequences of characters." [5]
 3. "Wonderful".length
 4. "Do spaces count?".length
2. The `length` method returns how many characters are in a string. However, the method will NOT give us the length of a number. If `num = 1001`, `num.length` returns `undefined` rather than 4.
 1. Use type conversion to print the length (number of digits) of an integer.
 2. Print the number of digits in a DECIMAL value (e.g. `num = 123.45` has 5 digits but a length of 6).
 3. What if `num` could be EITHER an integer or a decimal? Add an `if/else` statement so your code can handle both cases. (Hint: Consider the `indexOf()` or `includes()` string methods).

[Code it at repl.it](#)

3. Remember, strings are *immutable*. Consider a string that represents a strand of DNA: `dna = "TCG-TAC-gaC-TAC-CGT-CAG-ACT-TAa-CcA-GTC-cAt-AGA-GCT"`. There are some typos in the string that we would like to fix:
 1. Use the `trim()` method to remove the leading and trailing whitespace, and then print the results.
 2. Change all of the letters in the `dna` string to UPPERCASE and print the result.
 3. Note that if you try `console.log(dna)` after applying the methods, the original, flawed string is displayed. To fix this, you need to *reassign* the changes back to `dna`. Apply these fixes to your code so that `console.log(dna)` prints the DNA strand in UPPERCASE with no whitespace.

[Code it at repl.it](#)

4. Let's use string methods to do more work on the DNA strand:

1. Replace the gene 'GCT' with 'AGG', and then print the altered strand.
2. Look for the gene 'CAT' with `indexOf()`. If found print, 'CAT gene found', otherwise print, 'CAT gene NOT found'.
3. Use `slice()` to print out the fifth gene (set of 3 characters) from the DNA strand.
4. Use a template literal to print, "The DNA string is ___ characters long."
5. Just for fun, apply methods to `dna` and use another template literal to print, 'taco cat'.

[Code it at repl.it](#)

5. If we want to turn the string 'JavaScript' into 'JS', we might try `.remove()`. Unfortunately, there is no such method in JavaScript. However, we can use our cleverness to achieve the same result.
 1. Use string concatenation and two `slice()`'s to print 'JS' from 'JavaScript'.
 2. Without using `slice()`, use method chaining to accomplish the same thing.
 3. Use bracket notation and a template literal to print, "The abbreviation for 'JavaScript' is 'JS'."

4. Just for fun, try chaining 3 or more methods together, and then print the result.

[Code it at repl.it](#)

6. Some programming languages (like Python) include a `title()` method to return a string with Every Word Capitalized (e.g. `'title case'.title()` returns `Title Case`). JavaScript has no `title()` method, but that won't stop us! Use the string methods you know to print `'Title Case'` from the string `'title case'`.

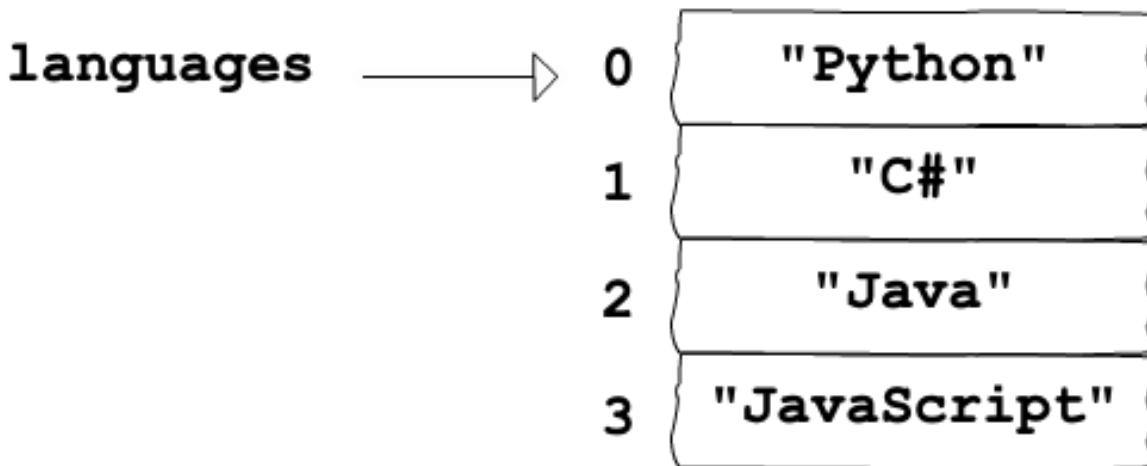
[Code it at repl.it](#)

ARRAYS KEEP THINGS IN ORDER

8.1 Arrays Are Like Strings

Arrays are similar to strings, but are a more general collection type. Like strings, **arrays** are a sequence of values that can be accessed via an ordered index. Unlike strings, arrays can store data of any type.

The figure below demonstrates an array named `languages`. The array contains four strings, each of those values has an index position.



8.1.1 Declaring an Array

Programmers use multiple ways to declare a new array. The simplest way is to use **array literal** notation `[]`. Anything enclosed in the square brackets will be *items* in the array. Each item should be followed by a comma `,`. If there are no items inside the brackets, then the array is considered empty.

```
1 let
2
3 let      "JavaScript"  "Python"  "Java"  "C#"
```

Array items can also be declared on multiple lines.

```
1 let
2   "React"
3   "Angular"
4   "Ember"
5   "Vue"
6
```

8.1.2 Array Length

To check the length of an array, use the `length` property, just like with strings. JavaScript array length is NOT fixed, meaning you can add or remove items dynamically.

Note

In other languages, such as Java and C#, arrays are of a static length requiring the length of the array to be declared upon creation.

Example

Print out the length of two arrays.

```
1 let
2
3
4 let           "JavaScript"  "Python"  "Java"  "C#"
5
```

Console Output

```
0
4
```

8.1.3 Varying Data Types

JavaScript arrays can hold a mixture of values of any type. For example, you can have an array that contains strings, numbers, and booleans.

```
let           "A string value"  true  99  105.5
```

Note

It's rare that you would store data of multiple types in the same array, because grouped data is usually the same type. In other languages, such as Java and C#, all items in an array have to be of the same type.

8.1.4 Check Your Understanding

Question

What is the length of the two arrays?

Hint: look closely at the quotes in the classes array.

```
1 let          "science, computer, art"  
2  
3 let          "Jones"  "Willoughby"  "Rhodes"
```

How can you change the `classes` array declaration to have the same number of items as the `teachers` array?

8.2 Working With Arrays

8.2.1 Bracket Notation and Index

As previously discussed, arrays are an ordered collection where each item can be accessed via index. Similar to strings, an **index** in an array is the number order given to items. Individual items can be accessed using bracket notation (`array[index]`). Indexes are zero-based, going from 0 to `array.length-1`.

Example

Use bracket notation and index to access items in an array.

```
1 let  
2   "JavaScript"  // index 0  
3   "Python"      // index 1  
4   "Java"        // index 2  
5   "C#"          // index 3  
6  
7           0  
8           3  
9  
10  // What will happen when index 4 is requested?  
11           4
```

Console Output

```
JavaScript  
C#  
undefined
```

Notice above that `undefined` was printed out when index 4 was referenced. `undefined` is returned when you request an index that the array does not contain.

Note

`undefined` is a special value in JavaScript that means no value has been assigned. We will discuss `undefined` more later in the class.

Example

`undefined` will be returned for any index that is outside of the array's index range.

```
1 let          "JavaScript"  "Python"   "Java"    "C#"
2               1
3               100
```

Console Output

```
undefined
undefined
```

8.2.2 Arrays are Mutable

In programming, mutability refers to what happens when you attempt to change a value. Remember that strings are immutable, meaning that any change to a string results in a new string being created. In contrast, arrays are **mutable**, meaning that individual items in an array can be edited without a new array being created.

Example

Update an item in an array using bracket notation and index.

```
1 let          "React"    "Angular"  "Ember"
2
3
4 // Set the value of index 2 to be "Vue"
5     2      "Vue"
6
7 // Notice the value at index 2 is now "Vue"
```

Console Output

```
[ 'React', 'Angular', 'Ember' ]
[ 'React', 'Angular', 'Vue' ]
```

8.3 Array Methods

As with strings, JavaScript provides us with useful **methods** for arrays. These methods will either *alter* an existing array, *return* information about the array, or *create and return* a new array.

8.3.1 Common Array Methods

Here is a sample of the most frequently used array methods. More complete lists can be found here:

1. [W3 Schools Array Methods](#)
2. [MDN Web Docs](#)

To see detailed examples for a particular method, control-click (or right-click) on its name.

Table 1: Methods That Return Information About The Array

Method	Syntax	Description
<i>includes</i>	arrayName. includes(item)	Checks if an array contains the specified item.
<i>indexOf</i>	arrayName. indexOf(item)	Returns the index of the FIRST occurrence of an item in the array. If the item is not in the array, -1 is returned.

Table 2: Methods That Rearrange The Entries In The Array

Method	Syntax	Description
<i>reverse</i>	arrayName.reverse()	Reverses the order of the elements in an array.
<i>sort</i>	arrayName.sort()	Arranges the elements of an array into increasing order (kinda).

Table 3: Methods That Add Or Remove Entries From An Array

Method	Syntax	Description
<i>pop</i>	arrayName.pop()	Removes and returns the LAST element in an array.
<i>push</i>	arrayName.push(item1, item2, ...)	Adds one or more items to the END of an array and returns the new length.
<i>shift</i>	arrayName.shift()	Removes and returns the FIRST element in an array.
<i>splice</i>	arrayName.splice(index, number, item1, item2, ...)	Adds, removes or replaces one or more elements anywhere in the array.
<i>unshift</i>	arrayName.unshift(item1, item2, ...)	Adds one or more items to the START of an array and returns the new length.

Table 4: Methods That Create New Arrays

Method	Syntax	Description
<i>concat</i>	arr.concat(otherArray1, otherArray2, ...)	Combines two or more arrays and returns the result as a new array.
<i>join</i>	arr.join('connecter')	Combines all the elements of an array into a string.
<i>slice</i>	arr.slice(start index, end index)	Copies selected entries of an array into a new array.
<i>split</i>	stringName.split('delimiter')	Divides a string into smaller pieces, which are stored in a new array.

8.3.2 Check Your Understanding

Follow the links in the table above for the *sort*, *slice*, *split* and *join* methods. Review the content and then answer the following questions.

Question

What is printed by the following code?

```

1 let      'coder'  'Tech'  47  23  350
2
3

```

- a. [350, 23, 47, 'Tech', 'coder']
- b. ['coder', 'Tech', 23, 47, 350]

- c. [23, 47, 350, 'coder', 'Tech']
 - d. [23, 350, 47, 'Tech', 'coder']
-

Question

Which statement converts the string `str = 'LaunchCode students rock!'` into the array `['LaunchCode', 'students', 'rock!']`?

- a. `str.join(" ")`;
 - b. `str.split(" ")`;
 - c. `str.join("")`;
 - d. `str.split("")`;
-

Question

What is printed by the following program?

```
1 let          'bananas'  'apples'  'edamame'  'chips'  'cucumbers'  'milk'  
2   ↪'cheese'  
3 let  
4  
5           2  5
```

- a. `['chips', 'cucumbers', 'edamame']`
 - b. `['chips', 'cucumbers', 'edamame', 'milk']`
 - c. `['cheese', 'chips', 'cucumbers']`
 - d. `['cheese', 'chips', 'cucumbers', 'edamame']`
-

8.4 Multi-Dimensional Arrays

Earlier we learned that arrays can store any type of value. If that is true, can we store arrays inside of arrays? Well yes we can....

A **multi-dimensional array** is an array of arrays, meaning that the values inside the array are also arrays. The *inner* arrays can store other values such as strings, numbers, or even more arrays.

The figure below demonstrates a `synonyms` array that has arrays as values. The *inner* arrays contain words that are synonyms of each other. Notice each inner array has an index position.

8.4.1 Two Dimensional Arrays

The simplest form of a multi-dimensional array is a two dimensional array. A two dimensional array is like a spreadsheet with rows and columns. To access items in a two dimensional array, use square bracket notation and two indexes `array[0][0]`. The first index is for the outer array, or the “row”, and second index is for the inner array, or the “column”.

	0	1	2
0	"table"	"grid"	"spreadsheet"
1	"determined"	"serious"	"strong"
2	"potential"	"possible"	"likely"
3	"enhance"	"improve"	"upgrade"

Note

The row and column analogy is used to help visualize a two dimensional array, however it's not a perfect analogy. There are no specific JavaScript language rules forcing the inner arrays to all have the same length. The inner arrays are separate arrays that can be of different length.

Example

Use a two dimensional array to contain three different lists of space shuttle crews.

```
1 let
2   'Robert Gibson'  'Mark Lee'  'Mae Jemison'
3   'Kent Rominger'  'Ellen Ochoa'  'Bernard Harris'
4   'Eileen Collins'  'Winston Scott'  'Catherin Coleman'
5
6
7   0  2
8   1  1
9   2  1
```

Console Output

```
Mae Jemison
Ellen Ochoa
Winston Scott
```

8.4.2 Multi-Dimensions and Array Methods

Both the inner and outer arrays in a multi-dimensional array are still altered with array methods.

Example

Use array methods to add an additional crew array and alter existing arrays.

```
1 let
2   'Robert Gibson'  'Mark Lee'  'Mae Jemison'
3   'Kent Rominger'  'Ellen Ochoa'  'Bernard Harris'
4   'Eileen Collins'  'Winston Scott'  'Catherin Coleman'
5
6
7 let           'Mark Polansky'  'Robert Curbbeam'  'Joan Higginbotham'
8
9 // Add a new crew array to the end of shuttleCrews
10
11           3  2
12
13 // Reverse the order of the crew at index 1
14           1
15           1
```

Console Output

```
Joan Higginbotham
[ 'Bernard Harris', 'Ellen Ochoa', 'Kent Rominger' ]
```

8.4.3 Beyond Two Dimensional Arrays

Generally there is no limit to how many dimensions you can have when creating arrays. However it is rare that you will use more than two dimensions. Later on in the class we will learn about more collection types that can handle complex problems beyond the scope of two dimensional arrays.

8.4.4 Check Your Understanding

Question

What are the two dimensional indexes for "Jones"?

```
1 let
2   "science"  "computer"  "art"
3   "Jones"    "Willoughby"  "Rhodes"
4
```

How would you add "dance" to the array at school[0]?

How would you add "Holmes" to the array at school[1]?

8.5 Exercises: Arrays

OK, rookie. It's time to train you on how to modify the shuttle's cargo manifest. The following actions will teach you how to add, remove, modify and rearrange our records for the items stored in our hold.

1. Create an array called `practiceFile` with the following entry: 273.15. Use the `push` method to add the following elements to the array. Add items a & b one at a time, then use a single `push` to add the items in part c. Print the array after each step to confirm the changes.

1. 42
2. "hello"
3. false, -4.6, "87"

[Code it at repl.it](#)

Congratulations, rookie. You can now add items to an array.

2. push, pop, shift and unshift are used to add/remove elements from the beginning/end of an array. **Bracket notation** can be used to modify any element within an array. Starting with the cargoHold array ['oxygen tanks', 'space suits', 'parrot', 'instruction manual', 'meal packs', 'slinky', 'security blanket'], write statements to do the following:

1. Use bracket notation to replace 'slinky' in the array with 'space tether'. Print the array to confirm the change.
2. Remove the last item from the array with pop. Print the element removed and the updated array.
3. Remove the first item from the array with shift. Print the element removed and the updated array.
4. Unlike pop and shift, push and unshift require arguments inside the (). Add the items 1138 and '20 meters' to the the array - the number at the start and the string at the end. Print the updated array to confirm the changes.
5. Use a template literal to print the final array and its length.

[Code it at repl.it](#)

Status check, rookie. Which array methods ADD items, and where are the new entries placed? Which methods REMOVE items, and where do the entries come from? Which methods require entries inside the “()”?

3. The splice method can be used to either add or remove items from an array. It can also accomplish both tasks at the same time. Review the *splice appendix* if you need a syntax reminder. Use splice to make the following changes to the final cargoHold array from exercise 2. Be sure to print the array after each step to confirm your updates.

1. Insert the string 'keys' at index 3 without replacing any other entries.
2. Remove 'instruction manual' from the array. (Hint: indexOf is helpful to avoid manually counting an index).
3. Replace the elements at indexes 2 - 4 with the items 'cat', 'fob', and 'string cheese'.

[Code it at repl.it](#)

Well done, cadet. Now let's look at some finer details about array methods. We've got to keep our paperwork straight, so you need to know when your actions change the original records.

4. Some methods—like splice and push—alter the original array, while others do not. Use the arrays

```
'duct tape'  'gum'  3.14  false  6.022e23
```

and

```
'orange drink'  'nerf toys'  'camera'  42  'parsnip'
```

to explore the following methods: concat, slice, reverse, sort. Refer back to the chapter if you need to review the proper syntax for any of these methods.

1. Print the result of using concat on the two arrays. Does concat alter the original arrays? Verify this by printing holdCabinet1 after using the method.

2. Print a `slice` of two elements from each array. Does `slice` alter the original arrays?
3. `reverse` the first array, and `sort` the second. What is the difference between these two methods? Do the methods alter the original arrays?

[Code it at repl.it](#)

Good progress, cadet. Here are two more methods for you to examine.

5. The `split` method converts a string into an array, while the `join` method does the opposite.
 1. Try it! Given the string `str = 'In space, no one can hear you code.'`, see what happens when you print `str.split()` vs. `str.split('e')` vs. `str.split(' ')` vs. `str.split('')`. What is the purpose of the parameter inside the `()`?
 2. Given the array `arr = ['B', 'n', 'n', 5]`, see what happens when you print `arr.join()` vs. `arr.join('a')` vs. `arr.join(' ')` vs. `arr.join('')`. What is the purpose of the parameter inside the `()`?
 3. Do `split` or `join` change the original string/array?
 4. The benefit, cadet, is that we can take a string with **delimiters** (like commas) and convert it into a modifiable array. *Try it!* Alphabetize these hold contents: “water,space suits,food,plasma sword,batteries”, and then combine them into a new string.

[Code it at repl.it](#)

Nicely done, astronaut. Now it's time to bring you fully up to speed.

6. Arrays can hold different data types, even other arrays! A **multi-dimensional array** is one with entries that are themselves arrays.
 1. Define and initialize the following arrays, which hold the name, chemical symbol and mass for different elements:
 - i. `element1 = ['hydrogen', 'H', 1.008]`
 - ii. `element2 = ['helium', 'He', 4.003]`
 - iii. `element26 = ['iron', 'Fe', 55.85]`
 2. Define the array `table`, and use `push(arrayName)` to add each of the element arrays to it. Print `table` to see its structure.
 3. Use bracket notation to examine the difference between printing `table[1]` and `table[1][1]`. Don't just nod your head! I want to HEAR you describe this difference. Go ahead, talk to your screen.
 4. Using bracket notation and the `table` array, print the mass of `element1`, the name for element 2 and the symbol for `element26`.
 5. `table` is an example of a *2-dimensional array*. The first “level” contains the element arrays, and the second level holds the name/symbol/mass values. **Experiment!** Create a 3-dimensional array and print out one entry from each level in the array.

[Code it at repl.it](#)

Excellent work, records keeper. Welcome aboard.

8.6 Studio: Strings and Arrays

Strings are **ordered collections** of *characters*, which are strings of length 1. The characters in a string can be accessed using **bracket notation**.

Arrays are ordered collections of items, which can be strings, numbers, other arrays, etc. The items/elements/entries stored in an array can be accessed using bracket notation.

Strings are **immutable**, whereas arrays can be changed.

Strings and arrays have **properties** and **methods** that allow us to easily perform some useful actions.

8.6.1 String Modification

Use string methods to convert a word into pseudo-pig latin.

- a. Remove the first three characters from a string and add them to the end. Ex: 'LaunchCode' becomes 'nchCodeLau'. Use a template literal to print the original and modified string in a descriptive phrase.
- b. Modify your code to accept user input. Query the user to enter the number of letters that will be relocated.
- c. Add validation to your code to deal with user inputs that are longer than the word. In such cases, default to moving 3 characters. Also, the template literal should note the error.

Code it at [repl.it](#)

8.6.2 Array and String Conversion

The `split` and `join` methods convert back and forth between strings and arrays. Use **delimiters** as reference points to split a string into an array, then modify the array and convert it back to a printable string.

- a. For a given string, use the `includes` method to check to see if the words are separated by commas (,), semicolons (;) or just spaces.
- b. If the string uses commas to separate the words, `split` it into an array, reverse the entries, and then `join` the array into a new comma separated string.
- c. If the string uses semicolons to separate the words, `split` it into an array, alphabetize the entries, and then `join` the array into a new comma separated string.
- d. If the string uses spaces to separate the words, `split` it into an array, reverse alphabetize the entries, and then `join` the array into a new space separated string.
- e. *Consider:* What if the string uses ‘comma spaces’ (,) to separate the list? Modify your code to produce the same result as part “b”, making sure that the extra spaces are NOT part of the final string.

Code it at [repl.it](#)

8.6.3 Bonus Mission: Multi-dimensional Arrays

Arrays can store other arrays!

- a. The cargo hold in our shuttle contains several smaller storage spaces. Use `split` to convert the following strings into four cabinet arrays. Alphabetize the contents of each cabinet.
 - i. “water bottles, meal packs, snacks, chocolate”
 - ii. “space suits, jet packs, tool belts, thermal detonators”
 - iii. “parrots, cats, moose, alien eggs”
 - iv. “blankets, pillows, eyepatches, alarm clocks”
- b. Initialize a `cargoHold` array and add the cabinet arrays to it. Print `cargoHold` to verify its structure.
- c. Query the user to select a cabinet (0-3) in the `cargoHold`.

- d. Use bracket notation and a template literal to display the contents of the selected cabinet. If the user entered an invalid number, print an error message instead.
- e. *Bonus to the Bonus:* Modify the code to query the user for BOTH a cabinet in `cargoHold` AND a particular item. Use the `includes` method to check if the cabinet contains the selected item, then print “Cabinet _____ DOES/DOES NOT contain _____.”

Code it at repl.it

REPEATING WITH LOOPS

9.1 Iteration

When repeating the same action over and over again, a human is likely to make a mistake. Computers, however, possess the incredible ability to carry out repetitive tasks without making mistakes.

To see this, let's consider an appropriate, if somewhat contrived, example. Suppose you want to print out the integers 0 through 50. With the tools you currently have at your disposal, your program would look like this:

```
1      0
2      1
3      2
4      3
5      4
6 // and so on...
```

Not only is this highly repetitive, but it is also error-prone. Even if utilizing copy-paste functionality, the sheer volume of code makes it somewhat likely that we will make a simple mistake, such as skipping an integer or misspelling `console`.

This code is also hard to modify. If we want to make a conceptually simple change—such as printing all the way to 100, or only printing even numbers—then we are forced to update an immense amount of code. Programming languages provide tools that allow us to repeat a sequence of statements in a much simpler way.

Repeated execution of a sequence of statements is called **iteration**. This chapter explores two mechanisms that JavaScript provides to make iteration simple and flexible—the `for` and `while` loops.

To give you a taste of what's to come, here is how we could write the program above using a `for` loop.

```
1 for let 0 51
2
3
```

We will explore the details of this syntax shortly, but it's worth taking a moment to marvel at the simplicity of this program compared to the one above.

Note

It may seem odd to you that this loop uses the integer 51, but only prints up to 50. Why this is the case will become clear in the next section.

Learning about iteration using loops is also an opportunity to introduce one of the most widely-known mnemonic devices in programming: *Don't Repeat Yourself*, or **DRY**. A common piece of advice from instructors and experienced programmers is that you should “keep your code DRY.” Let's learn how.

9.2 for Loops

The `for` loop is the first JavaScript tool for iteration that we will explore. A **for loop** is typically used for **definite iteration**. Definite iteration is the process of repeating a specific task with a specific data set. When a `for` loop begins it can usually be determined exactly how many times it will execute: once for each item in the data set.

9.2.1 for Loop Syntax

We have already seen the basic syntax of a `for` loop.

```
1 for let 0 51  
2  
3
```

This program prints the integers 0 through 50, one number per line. In the language of definite iteration, we say that the loop has a data set of 0-50, and its action is to print a value to the console.

Let's break down this syntax piece by piece, so we can begin to understand how `for` loops are structured.

A `for` loop always contains the following components:

```
for (initial expression; loop condition; update expression) {  
    loop body  
}
```

Notice that in the first line, within parentheses, the components **initial expression**, **loop condition**, and **update expression** are separated by semicolons. Let's look at these components in detail.

1. The statement `let i = 0` is executed exactly once, at the *beginning* of loop execution. The variable `i` is the **loop variable**.
2. The boolean expression `i < 51` is the **loop condition**. This condition is evaluated before each loop iteration, or repetition.
 - a. If the condition is `true` then the loop executes again.
 - b. If the condition is `false` then the loop ceases execution, and the program moves on to the code below the loop.
3. The statement `i++` is the **update expression**. This expression is executed at the *end* of each loop iteration.
4. The block of code surrounded with brackets (`{ }`) is the **loop body**. The body is executed once for each iteration of the loop.

9.2.2 Flow of Execution of the for Loop

In just a few lines of code, a `for` loop contains a lot of detailed logic, so let's spend some time breaking down the flow of execution for the particular loop that we've been looking at.

```
1 for let 0 51  
2  
3
```

Here is a step-by-step description of how this loop executes:

1. When the program reaches the `for` loop, the initial expression `let i = 0` is executed, declaring the variable `i` and initializing it to the value 0.

2. The loop condition `i < 51` is evaluated, returning `true` because 0 is less than 51.
3. Since the condition is `true`, the loop body executes, printing 0.
4. After the execution of the loop body, the update expression `i++` is executed, setting `i` to 1. This completes the first iteration of the loop.
5. Steps 2 through 4 are repeated, using the new value of `i`. This continues until the loop condition evaluates to `false` in step 2, ending the loop. In this example, this occurs when `i < 51` is `false` for the first time. Since our update expression adds 1 after each iteration, this occurs when `i` is 51 (so `51 < 51` is `false`). At that point, the loop body will have executed exactly 51 times, with `i` having the values 0...50.

In general, we can visualize the flow of execution of a `for` loop as a flowchart.

9.3 Iterating Over Collections

One of the most common uses of a `for` loop is to carry out a task once for each item in a collection. We have learned about two types of collections, strings and arrays. When using a loop with a collection in this way, we say that the loop *iterates over* the collection.

9.3.1 Iterating Over Strings

The following example prints each of the characters of the string "LaunchCode" on a separate line.

Example

```
1 let      "LaunchCode"
2
3 for  let    0
4
5
```

Console Output

```
L
a
u
n
c
h
C
o
d
e
```

Since `name.length` is 10, the loop executes once each for the values of `i` from 0 to 9. The loop body, `console.log(name[i]);`, will print `name[i]` each time. In each case, `name[i]` is one of the characters of `name`.

Try It!

Write a program that prints each character of your name on a different line.

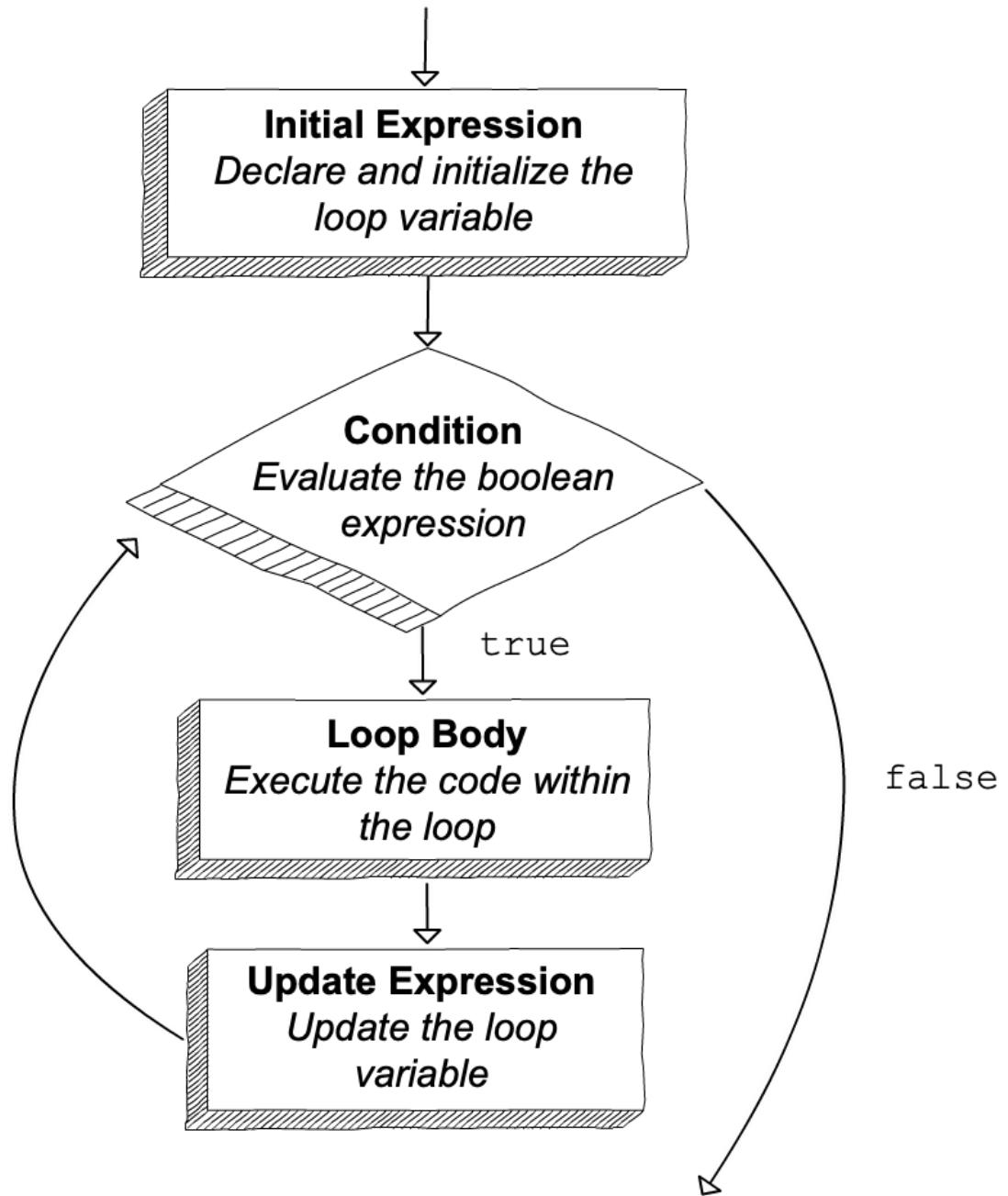


Fig. 1: Flow of execution of a `for` loop

```
1 // create a string variable containing your name  
2  
3 // write a for loop that prints each character in your name on a different line
```

repl.it

9.3.2 Iterating Over Arrays

The following example prints each of the programming languages in the array `languages` on a separate line.

Example

```
1 let languages = ["JS", "Java", "C#", "Python"]  
2  
3 for let i = 0  
4  
5
```

Console Output

```
JS  
Java  
C#  
Python
```

Similar to the string example, this loop executes 4 times because `languages.length` is 4. For each iteration, `languages[i]` is one of the items in the array and the given language is printed.

Try It!

Write a program that prints the name of each member of your family on a different line.

```
1 // create an array variable containing the names  
2  
3 // write a for loop that prints each name on a different line
```

repl.it

9.4 Breaking Down the `for` Statement

Having seen several examples, we will now explore the syntax of a `for` loop in more depth.

Recall the first example of a `for` loop that we looked at.

```
1 for let i = 0 to 51  
2  
3
```

We broke down the flow of execution of this loop, noting that the loop executes once for each of the values of `i` from `0...50`. The three components of the loop—loop variable, loop condition, and update expression—dictate exactly how this loop executes. So far, we have only seen `for` loops with this exact form:

```
1 for let 0
2   // loop body
3 }
```

However, the three components of a `for` loop statement can take different forms to create more complex looping behavior.

9.4.1 for Loop Anatomy

The general form of a `for` loop is:

```
for (initial expression; loop condition; update expression) {
  loop body
}
```

Let's look at each of the three components that affect how this loop iterates.

Initial Expression

The **initial expression** is executed once, before any iterations of the loop. It can be any expression, even the **empty expression** (which contains no code). However, it almost always declares and initializes a variable, known as the **loop variable**.

The loop variable can be initialized to any value.

Examples

This loop prints `3...9`.

```
1 for let 3 10
2
3 }
```

This loop prints each of the letters `C, o, d, and e` on a separate line.

```
1 let "LaunchCode"
2
3 for let 6
4
5 }
```

To avoid confusion and bugs, you should give your loop variable a unique name, one that you have not used elsewhere in your program. In cases where the loop variable is serving as a “counter” for iterations of a loop, it is conventional to use `i` for the variable name. In the case of nested `for` loops (loops inside of loops), the variables `j, k, etc.` are often used.

Note

The loop variable is typically used by the loop body, but this is not required. The following example is a valid `for` loop that prints `"LaunchCode"` 42 times.

```
1 for let 0 42
2           "LaunchCode"
3
```

Loop Condition

The **loop condition** is executed before each loop iteration. It is *always* a boolean expression, evaluating to `true` or `false`. If the condition is true, the loop body executes. If the condition is false, loop execution stops and the program continues with the next line of code below the loop.

Example

This loop does not iterate at all, because its condition is false to start with.

```
1 for let 0 1
2           "LaunchCode"
3
```

It is critical that the loop condition *eventually* becomes false. A loop for which the condition is never false is known as an **infinite loop**, because it never stops iterating. A program that contains an infinite loop will only stop after running out of memory or being manually stopped (for example, using control+c in a terminal).

Example

This is an infinite loop, because its condition will always be true.

```
1 for let 0 1
2           "LaunchCode"
3
```

You will accidentally write an infinite loop at some point; doing so is a rite of passage for new programmers. When this happens, don't panic. Stop your program and figure out why your loop condition never became false.

Update Expression

The final component in a for loop definition is the **update expression**, which executes after *every* iteration of the loop. While this expression may be anything, it most often updates the value of the loop variable.

In all of the examples we have seen so far, the update expression has been `i++`, incrementing the loop variable by 1. However, it can update the loop variable in other ways.

Example

This loop prints *even* integers from 0...50.

```
1 for let 0 51 2
2
3
```

A bad choice of update expression can also cause an infinite loop.

Example

This loop repeats indefinitely, since `i` becomes smaller with each iteration and thus is never greater than or equal to 51.

```
1 for let 0 51  
2  
3
```

repl.it

Try It!

How does each of these three components affect the behavior of a `for` loop? Experiment by modifying each of them in our example: the variable initialization, the boolean condition, and the update expression.

```
1 for let 0 51  
2  
3
```

9.4.2 Check Your Understanding

Consider the program:

```
1 let "LaunchCode's LC101"  
2  
3 for let 0 1 3  
4  
5
```

Question

How many times does the loop body execute?

1. 5
 2. 6
 3. 17
 4. 18
-

Question

Which set of characters is printed by the loop? (We have placed characters for the choices below on the same line, but they would be on separate lines in the actual program output.)

1. 'LaunchCode's LC101'
2. 'LaunchCode's LC10'

3. 'LnCe 1'
 4. 'LnCe '
-

9.5 The Accumulator Pattern

A **pattern** is a commonly-used approach to solve a group of similar programming problems.

This section introduces your first pattern, which we will explore in-depth after looking at a motivating example.

9.5.1 Adding 1...n

Let's write a program that adds up the integers 1...n, where n is an integer variable that we will create.

If you were to do this with pen and paper, you would write out a single formula and compute the answer. For example, for n = 6 you would write:

```
1 + 2 + 3 + 4 + 5 + 6
```

To get the result, you would first add 1 and 2 to get 3. Then you would add 3 and 3 to get 6. Then you would add 6 and 4 to get 10, and so on. The final result is 21.

We can carry out this same procedure in code using a loop.

Example

```
1 let      6
2 let      0
3
4 for let    1
5
6
7
8
```

Console Output

```
21
```

The variable `total` is initialized to 0. The loop executes once each for the values of `i` from 1 to 6. Each time the loop body executes, the next value of `i` is added to `total`.

The loop carries out the same basic algorithm that we used to compute the sum $1 + 2 + 3 + 4 + 5 + 6$ by hand. The only step that may seem different to you is the use of the variable `total` to keep track of the running total. When calculating the sum using pen and paper, we rarely write down this part, keeping track of the running total in our head. With programming, however, we must explicitly store such a value in a variable.

This pattern of initializing a variable to some basic, or empty value, and updating it within a loop is commonly referred to as the **accumulator pattern**. We refer to the variable as the **accumulator**. In the example above, `total` is the accumulator, and it “accumulates” the individual integers one by one.

The accumulator pattern comes up regularly in programming. The key to using it successfully is to initialize the accumulator variable before you start the iteration. Once inside the loop, update the accumulator.

9.5.2 Reversing a String

While some programming languages have a string method that will reverse a given string, JavaScript does not. Let's see how we can write our own program that reverses a string using the accumulator pattern.

We'll start by initializing two variables: the string we want to reverse, and a variable that will eventually store the reversed value of the given string.

```
1 let      "accumulator"
2 let      ""
```

Here, `reversed` is our accumulator variable. Our approach to reversing the string will be to loop over `str`, adding each subsequent character to the *beginning* of `reversed`, so that the first character becomes the last, and the last character becomes the first.

Example

```
1 let      "blue"
2 let      ""
3
4 for let    0
5
6
7
8
```

Console Output

```
eulb
```

Notice that we don't use the `+=` operator within the loop, since `reversed += str[i]` is the same as `reversed = reversed + str[i]`.

Let's break this down step-by-step. This table shows the values of each of our variables *after* each loop iteration.

Table 1: The accumulator pattern, step by step

Loop iteration (before first iteration)	i	str[i]	reversed
	not defined	not defined	""
1	0	"b"	"b"
2	1	"l"	"lb"
3	2	"u"	"ulb"
4	3	"e"	"eulb"

Try It!

What happens if you reverse the order of the assignment statement within the `for` loop, so that `reversed = reversed + str[i];`?

Try it at repl.it.

9.5.3 Summing an Array

Another common use of the accumulator pattern is to compute some value using each of the elements of an array. This is similar to adding $1\dots n$ as we did above, with the difference being we will use the items in an array rather than $1\dots n$.

Example

```
1 let      2   5   13  42
2 let      0
3
4 for  let  0
5
6
```

Console Output

```
52
```

9.6 while Loops

There is another JavaScript construct that can also be used for iteration, the `while` loop. The `while` loop provides a much more general mechanism for iterating. Like a `for` loop, it uses a condition to determine whether the loop body will continue to execute. Unlike a `for` loop, however, it does not have initial and update expressions.

9.6.1 while Loop Syntax

The general syntax of a `while` loop looks like this:

```
while (boolean expression) {
    body
}
```

A `while` loop will continue to repeat as long as its boolean expression evaluates to `true`. The condition typically includes a value or variable that is updated within the loop, so that the expression eventually becomes false.

9.6.2 Flow of Execution of the while Loop

We can visualize the flow of execution of a `while` loop as follows.

Here is the flow of execution for a `while` loop:

1. Evaluate the condition, which yields a value of `true` or `false`.
2. If the condition is `false`, exit the `while` loop and continue execution at the next statement after the loop body.
3. If the condition is `true`, execute the loop body and then go back to step 1.

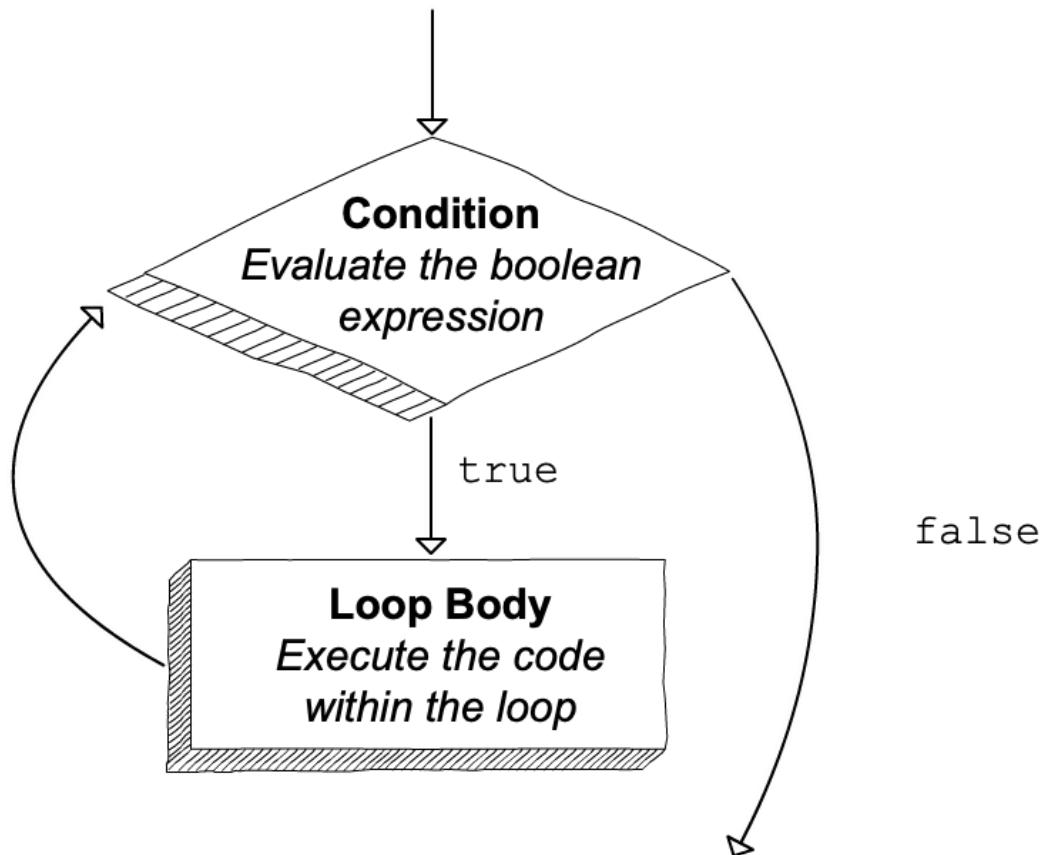


Fig. 2: Flow of execution of a `while` loop

9.6.3 for Loops Rewritten as while Loops

We can use the `while` loop to create any type of iteration we wish, including anything that we have previously done with a `for` loop. For example, consider our initial `for` loop example.

```
1 for  let      0      51
2
3
```

This can be rewritten as a `while` loop:

```
1 let      0
2
3 while      51
4
5
6
```

[repl.it](#)

Instead of relying on the the initial and update expressions, as we do in a `for` loop, we must manage the state of our loop manually. To do this, *before* entering the `while` loop, we will create the variable `i` and initialize it to 0, the first number we want to print. This variable plays the same role as the loop variable in a `for` loop. Every iteration will print `i` and then increment `i` to the next value, until it reaches the value 51. The loop continues to iterate until the condition `i < 51` evaluates to `false`.

You can almost read the `while` statement as if it were in a natural language: *while “i” is less than “51”, continue executing the body of the loop.*

Try It!

What happens if you forget to include `i++` at the end of the `while` loop above?

9.6.4 Beyond for Loops

We stated earlier that `while` loops are more flexible than `for` loops. Now we will look at an example that illustrates this.

This program is an example of **input validation**. It prompts the user to enter a positive number, converting the input string to the number data type. If the number is not positive, then the user is prompted again within the body of the loop. As long as the user continues to input non-positive numbers, the loop will continue to iterate.

```
1 const          'readline-sync'
2
3 let           'Please enter a positive number:'
4
5
6 while        0
7           'Invalid input. Please enter a positive number:'
8
9
```

This example illustrates the additional flexibility provided by `while` loops. While we use `for` loops to iterate over fixed collections (a string, an array, a collection of integers), the `while` loop can be used to iterate in more general circumstances. For the input validation example, at runtime it cannot be determined how many times the loop will repeat.

9.6.5 Infinite Loops, Revisited

It is easier to create an infinite `while` loop than an infinite `for` loop. To see this, consider what happens to our first `while` loop example if we forget to update the loop variable.

```
1 let      0
2
3 while      51
4
5
```

This is an infinite loop. The variable `i` is initialized to 0 and never updated, so the condition `i < 51` will always be true. If you ran this program, you would see an ever-increasing list of numbers.

9.6.6 Check Your Understanding

Question

You can rewrite any `for` loop as a `while` loop.

1. True
 2. False
-

Question

The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

```
1 let      10
2 let      1
3
4 while      0
5
6     1
7
8
9
```

1. `n` starts at 10 and is incremented by 1 each time through the loop, so it will always be positive.
 2. `answer` starts at 1 and is incremented by `n` each time, so it will always be positive
 3. You cannot compare `n` to 0 in a `while` loop. You must compare it to another variable.
 4. In the `while` loop body, we must set `n` to `false`, and this code does not do that.
-

9.7 Terminating a Loop With `break`

JavaScript, like most programming languages, provides a mechanism for terminating a loop before it would complete otherwise. The `break` keyword, when used within a loop, will immediately terminate the execution of any loop. Program execution then continues at the next line of code below the loop.

Example

This loop executes 12 times, for values of `i` from 0 to 11. During the twelfth iteration, `i` is 11 and the condition `i > 10` evaluates to `true` for the first time and execution reaches the `break` statement. The loop is immediately terminated at that point.

```

1 for let 0 42
2
3 // rest of loop body
4
5 if 10
6   break
7
8
9

```

The `break` statement can also be used within a `while` loop. Consider a situation where we are searching for a particular element in an array. (We have seen that JavaScript has array methods that can carry out array searches, but many programming languages do not.)

We can use a `while` loop to say, *while we have not reached the end of the array, continue iterating*. We can then include a `break` within a conditional check to say, *when we have found the element we are searching for, exit the loop*.

Example

A `while` loop can be used with `break` to search for an element in an array.

```

1 let      /* some numbers */
2 let      42
3 let      0
4
5 while
6   if
7     break
8
9
10
11
12 if
13   "The value"      "was located at index"
14 else
15   "The value"      "is not in the array."
16

```

Notice that we use a `while` loop in this example, rather than a `for` loop. This is because our loop variable, `i`, is used outside the loop. When we use a `for` loop in the way we have been, the loop variable exists only within the loop.

9.8 Choosing Which Loop to Use

The `for` loop is typically used to iterate through a fixed set of values that can be determined before the loop executes. This is why we say that a `for` loop exhibits **definite iteration**.

On the other hand, the `while` loop is more flexible, as we saw with the example of validating user input. In that case, we could not determine in advance how many times the loop would iterate; it depended entirely on the values provided by the user during program execution. For this reason, a `while` loop is often described as **indefinite iteration**. We

expect that *eventually* the condition controlling the iteration will evaluate to `false` and the iteration will stop. (Unless we have an infinite loop, which is a problem we want to avoid.)

While we saw that any `for` loop can be written as a `while` loop by manually creating and updating a loop variable, it is preferable to use a `for` loop when iterating over a collection or iterating a fixed number of times. Manually updating the variable in a `while` loop is more work for you, the programmer, and can lead to infinite loops if not handled properly.

9.8.1 Check Your Understanding

Question

You are asked to program a robot to move tennis balls from one box (Box #1) to another (Box #2), one-by-one. The robot should continue moving balls until Box #1 is empty, and balls may be added to the box after the robot begins its work.

Which type of loop should you use to write the program?

1. `while` loop
 2. `for` loop
-

Question

You are asked to write a program similar to the one above, with the modification that a user may give the robot a specific number of balls to move from Box #1 to Box #2. (You can assume there will always be more balls than the user has asked the robot to move.)

Which type of loop should you use to write the program?

1. `while` loop
 2. `for` loop
-

9.9 Exercises: Loops

Practice makes better. Repetition is a good thing.

WAIT!!! Why type “Repetition is a good thing,” four times when we can code a better result? How about printing the phrase 100 times instead?

```
1  for  let      0      100
2          "Repetition is a good thing."  
3
```

Loops simplify repetitive tasks!

9.9.1 `for` Practice

[Code it at repl.it](#)

1. Construct `for` loops that accomplish the following tasks:
 - a. Print the numbers 0 - 20, one number per line.
 - b. Print only the ODD values from 3 - 29, one number per line.
 - c. Print the EVEN numbers 12 down to -14 in descending order, one number per line.
 - d. Print the numbers 50 down to 20 in descending order, but only if the numbers are multiples of 3.
2. Initialize two variables to hold the string 'LaunchCode' and the array [1, 5, 'LC101', 'blue', 42], then construct `for` loops to accomplish the following tasks:
 - a. Print each element of the array to a new line.
 - b. Print each character of the string—in reverse order—to a new line.
3. Construct a `for` loop that sorts the array [2, 3, 13, 18, -5, 38, -10, 11, 0, 104] into two new arrays:
 - a. Define a `evens` array for the evens and an `odds` array for the odds.
 - b. Loop over the numbers and put them into the correct array based on their even/odd status.
 - c. Print the arrays to confirm the results, print `evens` first. Example: `console.log(evens);`

9.9.2 `while` Practice

[Code it at repl.it](#)

Define three variables for the LaunchCode shuttle—one for the starting fuel level, another for the number of astronauts aboard, and the third for the altitude the shuttle reaches.

4. Construct `while` loops to do the following:
 - a. Prompt the user to enter the starting fuel level. The loop should continue until the user enters a positive value greater than 5000 but less than 30000.
 - b. Use a second loop to query the user for the number of astronauts (up to a maximum of 7). Validate the entry by having the loop continue until the user enters an integer from 1 - 7.
 - c. Use a final loop to monitor the fuel status and the altitude of the shuttle. Each iteration, decrease the fuel level by 100 units for each astronaut aboard. Also, increase the altitude by 50 kilometers.
5. After the loops complete, output the result with the phrase, "The shuttle gained an altitude of ___ km."
 1. If the altitude is 2000 km or higher, add "Orbit achieved!"
 2. Otherwise add, "Failed to reach orbit."

9.10 Studio: Loops

Now that we've launched our shuttle, let's use loops (iteration) to automate some tasks.

9.10.1 Part A (Put dinner together)

1. First, initialize variables to store the following arrays. Remember to use descriptive names.

- Protein options:

```
['chicken', 'pork', 'tofu', 'beef', 'fish', 'beans']
```

- Grain options:

```
['rice', 'pasta', 'corn', 'potato', 'quinoa', 'crackers']
```

- Vegetable options:

```
['peas', 'green beans', 'kale', 'edamame', 'broccoli', 'asparagus']
```

- Beverage options:

```
['juice', 'milk', 'water', 'soy milk', 'soda', 'tea']
```

- Dessert options

```
['apple', 'banana', 'more kale', 'ice cream', 'chocolate', 'kiwi']
```

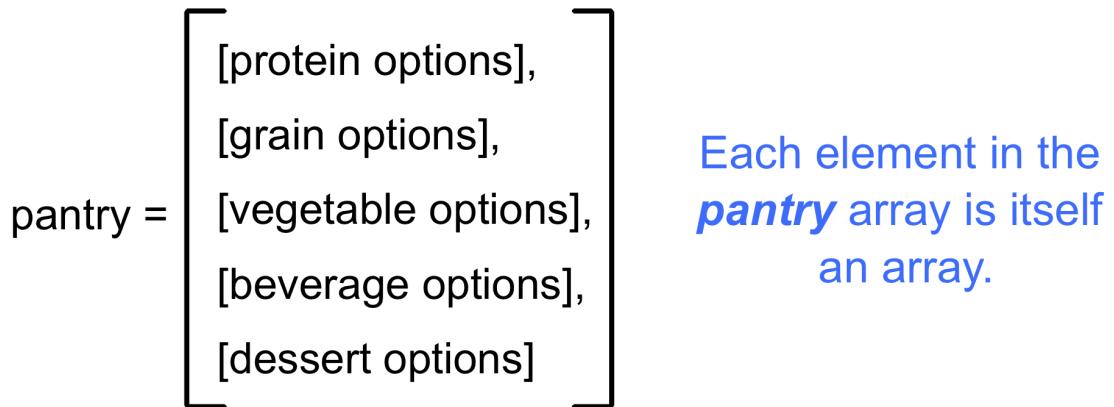
Code it at repl.it

2. Use a `for` loop to assemble 6 meals.

- a. The meals must include one item from each of the source arrays.
- b. Each ingredient can only be used ONCE.
- c. Print out each meal.

3. *Skill boost!* (Optional): To enhance your learning, modify your code to:

- a. Use string formatting to print something more interesting than “[‘chicken’, ‘rice’, ‘peas’, ‘juice’, ‘apple’]” for the meal outputs.
- b. Use an “array of arrays” to store the food options in a ‘pantry’.



```
pantry = [['chicken', 'pork', ...], ['rice', ...], ['peas', ...], ['juice', ...], ['apple', ...]]
```

9.10.2 Part B (Self-destruct system)

If the shuttle gets hijacked by space pirates, the astronauts can activate a self-destruct sequence to provide some drama for the viewers at home.

In order to prevent a rogue astronaut from activating the code, it takes *two* crew members to begin the countdown. Each person must enter a different code, after which the computer will “zip” them together before overloading the engines.

4. Construct a `for` loop that combines two strings together, alternating the characters from each source.
-

Examples

1. If `string = "1234"` and `otherString = "5678"`, then the output will be “15263748”.
 2. If `code1 = "ABCDEF"` and `code2 = "notyet"`, then the output will be “AnBoCtDyEeFt”.
 3. If `ka = "LoOt"` and `blam = "oku!"`, then the output will be “LookOut! ”.
-

[Code it at repl.it](#)

9.10.3 Part C (Refinements)

Update your code from part A to add user input and validation.

5. Using a `while` loop, ask the user to select the number of meals to assemble. Validate the input to make sure it is an integer from 1 - 6.

9.10.4 Bonus Mission

Modify your code to check each meal for kale. If present, after the meal output add, “Don’t worry, you can have double chocolate tomorrow.”

FUNCTIONS ARE AT YOUR BECK AND CALL

10.1 Introduction

You have been using functions throughout your learning so far, without receiving a full explanation of how functions work. This chapter focuses explicitly on the details of how functions work, how they can be used, and how you can create functions of your own.

A **function** is a reusable, callable piece of code. Like variables, functions often have names (though the next chapter shows us that we can create functions without names).

You have already become familiar with several functions:

- `console.log`
- The type conversion functions: `Number`, `String`, and `Boolean`
- String and array methods, such as `indexOf`

Note

When learning about strings and arrays, we noted that a **method** is a function that “belongs to” an object. This distinction is important to keep in mind, and will be explored in depth in a later chapter. For now, think of a method as a *special type* of function.

Each of the functions we have used works in the same way. By typing the function’s name, followed by parentheses, we can *call* the function, resulting in an action being carried out. Sometimes, as with `console.log`, we can provide input data between the parentheses, which the function will use to carry out its action.

Example

The function `console.log` prints the provided value or values (the input data).

```
"Hello, World!"
```

Console Output

```
Hello, World!
```

This is an example of a function receiving *input*. Functions may also provide *output*. For example, the type conversion functions give back the result of converting a value.

Example

Introduction to Professional Web Development in JavaScript

Type conversion functions *return* a value, that can be used by the calling code. Often, we store the return value of a function in a variable.

```
1 let num = "42"
2   "The variable num is of type" typeof num "and has value"
```

Console Output

```
The variable num is of type number and has value 42
```

Example

Many array and string methods also return values. This program uses the string method `split` to break a string into separate components.

```
1 let address = "Smith, Jane, 100 Cherry Blossom Lane"
2 let arr = address
3   ','
```

Console Output

```
[ 'Smith', 'Jane', '100 Cherry Blossom Lane' ]
```

Functions are extremely powerful. They allow us to repeat actions without repeating each individual step of code that the actions are built from. By grouping actions together, functions allow us to be removed from the details of what they are actually doing.

When we want to print a message to the console using `console.log`, we don't have to know what the console is, or how a string can be displayed on it. The behavior is wrapped up within the function itself. This is an example of a broader programming concept known as **encapsulation**. Encapsulation is the process of packaging up code in a reusable way, without the programmer needing to be concerned with how it works.

A commonly-used analogy for describing the concept of a function is that of a machine that takes input, carries out an action, and gives back a result. This is known as the **function machine** analogy.



Fig. 1: The function machine

If we want to use a function, we must provide it with some input. It carries out an action on that input and returns a result. The action occurs within the function, or “inside the machine”. If we know the purpose of a function, we simply provide it with input and receive the output. The rest is up to the machine itself.

Note

You may notice that a function like `console.log` doesn't seem to return anything. We will soon learn that *every* function returns a value, regardless of whether or not that value is used, or is even useful.

The programming concept of a function is very similar to the concept of a mathematical function. For example, in high school algebra you learned about functions like $y = 4x + 7$. These functions used a mathematical input (x) and carried out a procedure to return a numerical result (y).

Example

Consider the following mathematical function:

$$f(x) = x^2 + 4x - 2$$

We can *call* the function by giving it a specific *input*:

$$f(3) = 3^2 + 4 \cdot 3 - 2 = 9 + 12 - 2 = 19$$

The number 19 is the *output*.

Functions also allow us to keep our code DRY, a concept that you learned about *when we introduced loops*. If we want to do the same basic task 17 times across a program, we can reduce code repetition by writing one function and calling it 17 times.

10.1.1 Check Your Understanding

Question

In your own words, explain what a function is.

10.2 Using Functions

Having informally used and discussed functions, it is time to formalize a few concepts.

A **function call** is the act of using a function by referring to its name, followed by parentheses. A synonymous term is **function invocation**, and we will sometimes say that we are “invoking a function.”

Within parentheses, a comma-separated list of **arguments** may be provided when calling a function. These are sometimes called **inputs**, and we say that the inputs are “passed to” the function.

A generic function call looks like this:

```
functionName(argument1, argument2, ..., argumentN);
```

Every function provides a **return value**, which can be used by the calling program—for example, to store in a variable or print to the console.

Example

A return value may be stored in a variable.

```
let
```

```
42
```

It may also be used in other ways. For example, here we use the return value as the input argument to `console.log` without storing it.

```
42
```

Console Output

```
42
```

If a function doesn't provide an explicit return value, the special value `undefined` will be returned.

Example

```
1 let          "LaunchCode"  
2
```

Console Output

```
LaunchCode  
undefined
```

Warning

The special value `undefined` is built into JavaScript. As with booleans, it is not a string, so `undefined === "undefined"` returns `false`.

In some cases, calling a function results in an action that changes the state of a program outside of the function itself. Such a behavior is known as a **side effect**.

Example

Calling `console.log` results in output to the console, which is a side effect.

10.3 Creating Functions

While using the functions built into JavaScript is useful, the most powerful aspect of functions is the ability of programmers to create their own.

There are several ways to define functions in JavaScript. We will introduce one technique in this chapter and a second technique in the next.

10.3.1 Function Syntax

To create a function, use the following syntax:

```

1 function
2
3 // function body
4
5

```

Here, `function` is a keyword that instructs JavaScript to create a new function using the definition that follows. Since `function` is a keyword, it may not be used elsewhere, for example as the name of a variable.

Following `function` is the **function name**, which is `myFunction` in the generic example above. The function name is determined by you, the programmer, and should therefore follow best practices. In particular, function names should use camel case and have descriptive names. We will have more to say about naming functions near the end of this chapter.

Following the function name, we define **parameters** within the parentheses. Think of parameters as variables that can be used only within the function itself. The number and names of the parameters are determined by the programmer, based on what they want the function to do. A function may be defined with several parameters, or no parameters at all.

Note

Many programming languages require you to state which data type each parameter should be (for example, string or number). In such languages, if you try to call a function with a parameter of incorrect type, an error results.

JavaScript *does not* allow you to specify the types of parameters, though the JavaScript extension TypeScript does. We will learn a bit of TypeScript later on.

After the parameters and closing parenthesis, within curly brackets, `{ }`, is the **function body**. This is where the actions that the function should carry out are defined. The function body can consist of any amount of code.

An Example

Let's see function syntax in action. We first consider a program that prints an array of names.

```

1 let      "Lena"  "James"  "Julio"
2
3 for  let    0
4
5

```

Following this pattern, we can create a function that prints *any* array of names.

```

1 function
2   for  let    0
3
4
5

```

Breaking down the components of a function using our new terminology gives us:

- **Function name:** `printNames`
- **Parameter(s):** `names`
- **Body:**

```
1 for let    0  
2  
3
```

Notice that there is nothing about this function that forces names to actually contain names, or even strings. The function will work the same for any array it is given. Therefore, a better name for this function would be `printArray`.

Our function can be used the same way as each of the built-in functions, such as `console.log`, by calling it. Remember that calling a function triggers its actions to be carried out.

```
1 function
2   for let    0
3
4
5
6
7     "Lena"  "James"  "Julio"
8     "___"
9     "orange" "apple"  "pear"
```

Console Output

```
Lena
James
Julio
---
orange
apple
pear
```

This example illustrates how functions allow us to make our code **abstract**. Abstraction is the process of taking something specific and making it more general. In this example, a loop that prints the contents of a specific array variable (something specific) is transformed into a function that prints the contents of *any* array (something general).

10.3.2 Defining and Calling

When we define a function, we are making it available for later use. The function does not execute when it is defined; it must be *called* in order to execute. This is not only a common point of confusion for new programmers, but can also be the source of logic errors in programs.

Let's see how this works explicitly.

Try It!

What happens if we define a function without calling it?

```
1 function
2   "Hello, World!"
```

repl.it

Question

What is printed when this program runs?

In order for a function to run, it must be explicitly *called*.

Example

```
1 function
2     "Hello, World!"
```

Console Output

```
Hello, World!
```

10.4 Function Input and Output

In the introduction to this chapter, we used the metaphor of the *function machine*, noting that the machine takes *input* and provides *output*. This section focuses on the details of these two aspects of function behavior.

10.4.1 Return Statements

Some functions return values that are useful. In particular, the type conversion functions convert input to the specified data type and return the result—calling `Number("3.14")` returns the value `3.14`.

Returning a Value

To return a value from functions that *we* create, we can use a **return statement**. A return statement has the form:

```
return
```

where `someVal` is any value.

Example

This function has a single parameter, `n`, which is expected to be a positive integer. It returns the sum `1+2+...+n`.

```
1 function
2     let      0
3     for let    0
4
5
6     return
7
8
9         3
```

Console Output

6

Notice that `sumToN` does not print anything; the output comes from the final line of the program, which prints the value *returned by* the function call `sumToN(3)`.

Now that we have `return` statements in our coding toolbox, we will very rarely print anything *within* a function. If we want to see the value returned by a function then we must print it *after* calling the function.

Question

The function `sumToN` uses a pattern that we have seen previously. What is it called?

Using `return` is Optional

As we saw with our initial examples of function definitions, not every function explicitly returns a value. At its simplest, a function can even have an empty body.

```
function
```

As written, this function is completely valid, but useless. Although the function doesn't have a `return` statement, JavaScript still implicitly returns a value.

Example

A function without a `return` statement returns the special value `undefined`.

```
1 function
2
3 let
4
```

Console Output

```
undefined
```

`return` Terminates Function Execution

When a `return` statement executes, the function terminates, regardless of whether or not there is any code following the `return` statement. This means that you must be careful to use `return` only when the work of the function has been completed.

Example

This `console.log` statement in this function never executes, since the function returns before it is reached.

```
1 function
2   return "I'm done!"
3     "This will not be printed"
4
5
6
```

Console Output

```
I'm done!
```

We can use the fact that `return` stops the execution of a function intentionally, to force a function to stop execution.

Example

This function prints out the integers 1...n using an infinite `while` loop, which nonetheless terminates when the `return` statement is executed.

```
1 function
2     let      1
3     while true
4         if
5             return
6
7
8
9
10
```

Boolean Functions

A function that returns a boolean value is known as a **boolean function**. Perhaps the simplest such function is one that tests an integer to determine if it is even.

Example

```
1 function
2     if      2      0
3         return true
4     else
5         return false
6
7
8
9             4
10            7
```

Console Output

```
true
false
```

It is conventional to name boolean functions by starting with either `is` or `has`, which creates a nice semantic effect when reading the code. For example, reading `isEven(4)` communicates to the reader that the function should answer the question, “Is 4 even?” This is a convention so widely used by programmers that it extends to nearly every language.

Let’s return to the `isEven` function above, to see how we can use the power of `return` statements to make it even better.

Since `return` terminates the function, we can leave out the `else` clause and have the same effect. This is because if `n` is even, the `return` statement in the `if` block will execute and the function will end. If `n` is odd, the `if` block will be skipped and the second `return` statement will execute.

```
1  function
2    if      2      0
3      return true
4
5  return false
6
```

This updated version works exactly the same as our initial function.

Additionally, notice that the function returns `true` when `n % 2 === 0` returns `true`, and it returns `false` when `n % 2 === 0` returns `false`. In other words, the return value is *exactly the same* as the value of `n % 2 === 0`. This means that we can simplify the function even further by returning the value of this expression.

```
1  function
2    return      2      0
3
```

This version of `isEven` is better than the first two, not because it is shorter (shorter isn't always better), but because it is simpler to read. We don't have to break down the conditional logic to see what is being returned.

Most boolean functions can be written so that they return the value of a boolean expression, rather than explicitly returning `true` or `false`.

10.4.2 Parameters and Arguments

Over the past few sections, we introduced two function-related concepts that are very similar, and are often confusing to distinguish: *arguments* and *parameters*. The difference between the two is subtle, so we will attempt to clear that up now.

The easiest way to talk about the difference between arguments and parameters is by referring to an example.

Example

The function `hello` takes a single value, which we expect to be a person's name, and returns a message that greets that person.

```
1  function
2    return `Hello, ${      }!`
3
4
5      "Lamar"
```

Console Output

```
Hello, Lamar!
```

In this example, `name` is a **parameter**. It is part of the function definition, and *behaves like a variable* that exists only within the function.

The value "`Lamar`" that is used when we invoke the function on line 5 is an **argument**. It is a *specific value* that is used during the function call.

The difference between a parameter and an argument is the same as that between a variable and a value. A variable *refers to* a specific value, just like a parameter *refers to* a specific argument when a function is called. Like a value, an argument is a concrete piece of data.

10.4.3 Arguments Are Optional

A function may be defined with several parameters, or with no parameters at all. Even if a function is defined with parameters, JavaScript will not complain if the function is called *without* specifying the value of each parameter.

Example

```
1 function
2   return `Hello, ${ }!`  
3
4
5
```

Console Output

```
Hello, undefined!
```

We defined `hello` to have one parameter, `name`. When calling it, however, we did not provide any arguments. Regardless, the program ran without error.

Arguments are optional when calling a function. When a function is called without specifying a full set of arguments, any parameters that are left without values will have the value `undefined`.

If your function will not work properly without one or more of its parameters defined, then you should define a **default value** for these parameters. The default value can be provided next to the parameter name, after `=`.

Example

This example modifies the `hello` function to use a default value for `name`. If `name` is not defined when `hello` is called, it will use the default value.

```
1 function          "World"
2   return `Hello, ${ }!`  
3
4
5
6   "Lamar"
```

Console Output

```
Hello, World!
Hello, Lamar!
```

While this may seem new, we have already seen a function that allows for some arguments to be omitted—the string method `slice`.

Example

The string method `slice` allows the second argument to be left off. When this happens, the method behaves as if the value of the second argument is the length of the string.

```
1 // returns "Launch"
2 "LaunchCode"      0  6
3
4 // returns "Code"
5 "LaunchCode"      6
6
7 // also returns "Code"
8 "LaunchCode"      6  10
```

Just as it is possible to call a function with *fewer* arguments than it has parameters, we can also call a function with *more* arguments than it has parameters. In this case, such parameters are not available as a named variable.

Example

This example calls `hello` with two arguments, even though it is defined with only one parameter.

```
1 function          "World"
2   return `Hello, ${      }!
3
4
5           "Jim"  "McKelvey"
```

Console Output

```
Hello, Jim!
```

Fun Fact

These “extra” arguments can still be accessed using a special object named `arguments`, which is made available to every function. If you are curious, [read more at MDN](#). However, we will not need to use this advanced JavaScript feature in this course.

10.4.4 Check Your Understanding

Question

What does the following code output?

```
1 function
2   return      2
3
4
5 let      2
6
7 for  let  0      4
8
9
10
11
```

Question

What does the following function *return*?

```
1  function
2    let
3
4
5
6    'Bob'
```

-
1. "BobBob"
 2. Nothing (no return value)
 3. undefined
 4. The value of Bob
-

Question

What does the following code *output*?

```
1  function
2    let
3
4
5
6    'Bob'
```

-
1. "BobBob"
 2. Nothing (no output)
 3. undefined
 4. The value of Bob
-

10.5 A Good Function-Writing Process

The function is the most complex JavaScript construct that we have seen. Functions have more components to their syntax than conditionals or loops, and can be used in more intricate ways than those constructs.

To avoid frustration and bugs, it's important to approach writing functions in an intentional, structured way. This is essential as you start to write more complex functions.

In this section, we outline what we think is the best approach. To provide concrete examples, we will consider a fictional function that is able to make a sandwich.

10.5.1 Step 1: Design Your Function

Before putting fingers to keyboard, it is important to have a clear idea of what you want your function to do. You should ask yourself the following questions:

- What data (that is, parameters) does my function need to do its job?

- Should my function return a value? (Hint: The answer is almost always “yes.”)
- What should be the data type of my function’s return value?
- What is a good, descriptive name for my function?
- What data types do we expect the parameters to be?
- What are good names for my parameters?

For our sandwich function, the answers might look like this:

Table 1: Specification for a Sandwich Function

Parameters	bread, filling, condiments
Return Value	The finished sandwich
Return Type	An object of type 'sandwich' *
Function name	makeSandwich
Parameter names and types	breadType (string), fillingType (string), condiments (array of strings)

* *JavaScript does not actually have a “sandwich” data type, but we want our function to be as flexible as possible. For now, recognize that returning a simple string to describe the sandwich will not be useful. In later lessons, we will learn how to create custom data types, so making a virtual, code-based “sandwich” here is not a problem.*

10.5.2 Step 2: Create the Basic Structure

Now it is time to start coding. Using the design decisions you just made, write the minimal syntax needed to create the function.

Here’s what an outline of our sandwich function would look like:

```
1 function
2
3 // TODO: make a sandwich with the given ingredients
4
5
```

Doing this step before writing the body will prevent silly mistakes like leaving off a } or forgetting to define a parameter.

10.5.3 Step 3: Write the Body

With the basic structure in place, go ahead and start writing the function body. Be sure to alternate between sub-tasks and running your code. *Do not wait until you have written the entire function body before testing your code!*

We can’t emphasize this enough. Going long stretches of time without running the program is a good way to end up frustrated. Recall in the chapter on debugging that we made the following recommendation to avoid bugs:

Get something working and keep it working.

This applies *especially* to writing functions. Every good professional programmer works in this way: write a few lines of code, run it, debug any errors, repeat.

Following these steps won’t prevent you from making mistakes, but it will certainly reduce the number of bugs you create. This helps you more quickly produce solid, working code.

10.6 Parameters and Variables

Earlier, we said that a parameter “behaves like a variable that exists only within the function.” While this is true, the relationship between variables and parameters is a bit more complicated.

10.6.1 Function Scope

The **scope** of a variable is the extent to which a variable is visible within a program. Scope consists of all locations within a program where a variable can be used or modified. Introducing functions gives us one of our first examples of limited variable scope—a situation in which a variable is not visible throughout an entire program.

In particular, *a variable defined using ‘let’ within a function is not visible outside of that function.*

Example

This function takes a string and returns the result of removing all hyphens, `-`, from the string.

```
1  function
2    let
3
4    for let      0
5      if        ' - '
6
7
8
9
10
11   let          "314-254-0107"
12
13
```

Console Output

```
3142540107
ReferenceError: strWithoutHyphens is not defined
(rest of error message omitted)
```

The last line of this program tries to print the variable `strWithoutHyphens` to the console, resulting in an error. The previous line calls `removeHyphens`, at the end of which `strWithoutHyphens` has the value `"3142540107"`. However, once the function finishes execution all variables and parameters within the function are destroyed. This is why the last line results in a `ReferenceError`; there is no variable named `strWithoutHyphens` in existence when that line executes.

This is what we mean when we refer to scope. A variable is not necessarily usable throughout an entire program. Where it can be used depends on the context in which it is defined. For variables *and* parameters within a function, their scope is known as **function scope**. This means that they are only visible within the function in which they are defined.

10.6.2 Variable Shadowing

We just learned that variables and parameters defined within a function are not visible outside of that function. The opposite scenario is more complicated; a variable defined outside a function *may* be visible within the function, in certain circumstances.

Example

In some cases, a variable defined outside of a function may be visible within the function.

```
1 let      "Hello, World!"  
2  
3 function  
4  
5  
6  
7
```

Console Output

```
Hello, World!
```

Even though message is defined outside the function, it is still visible within the function. When printMessage is called and `console.log(message);` executes, message has the value "Hello, World!", so that value is printed to the console. This means that the scope of message extends to the function printMessage.

Warning

It is NOT the case that all variables defined outside of a function are visible within *every* function. The reality is a bit more nuanced than this, and will be explored in depth in a later chapter.

Try It!

What is the output of the following program? Form a hypothesis for yourself before running it.

Once you have answered that question, try relocating the declaring message to other locations to see how it affects the program. For example, you might try placing it within or after printMessage.

```
1 let      "Hello, World!"  
2  
3 function  
4  
5  
6  
7  
8   "Goodbye"  
9
```

repl.it

An interesting thing happens when a function parameter has the same name as a variable that is in-scope.

Example

```
1 let      "Hello, World!"  
2  
3 function  
4  
5
```

(continues on next page)

(continued from previous page)

```
6           "Goodbye"  
7
```

Console Output

```
Goodbye
```

While the variable `message` declared on line 1 is technically visible within `printMessage` (that is, it is in-scope), it is hidden by the function parameter of the same name. When `printMessage("Goodbye")` is called and `console.log(message)` executes, `message` has the value "Goodbye", which is the argument passed into the function. This phenomenon is known **shadowing**, based on the metaphor that a function parameter “casts its shadow over” a variable of the same name, effectively hiding it.

There is no good reason to intentionally use variable shadowing in your program. In fact, doing so can lead to confusion over which of the two variables is being used in a given situation. For this reason, *you should avoid naming variables and function parameters the same name*.

10.6.3 Check Your Understanding

Question

What does the following code output?

```
1 let      42  
2  
3 function  
4   return    2    0  
5  
6  
7           43
```

10.7 Naming Functions

As with variables, choosing good, descriptive names for the functions you write is important. It makes your code more readable, and therefore more maintainable and more bug-resistant.

10.7.1 Use Camel Case

As with variables, use camel case. All functions in JavaScript should begin with a lowercase letter, with the first letter of subsequent words capitalized.

Example

Good

- `const astronautCount = 7;`
- `const fuelTempCelsius = -225;`
- `let isReady = false;`

Bad

- const AstronautCount = 7;
 - const fuel_temp_celsius = -225;
 - let is_ready = false;
-

10.7.2 Use Verb/Noun Pairs When Applicable

A function carries out an action, and it often produces some specific output or effect. Therefore, using a verb/noun pairs can go a long way toward making it clear what a functions does. A good verb can describe the action, and a good noun can describe the output, or the object that is being affected by the function.

Example

Good

- prepareForLiftoff
- fillFuelTank
- getCountdownStatus
- isReadyForLiftoff

Bad

- liftoff
 - fillup
 - countdownStatus
 - isReady
-

As we noted earlier, for boolean functions it is conventional that their names start with “is” or “has” whenever possible. Creating a verb/noun pair for a function name doesn’t always make sense, but when it does, you should use this format to create a good, descriptive name.

10.7.3 Use Descriptive Names

We have repeatedly reminded you to use descriptive names, but now we want to expand on this point. You should *prefer long, descriptive names over short, abbreviated names*. If you can read a function name and understand what it does from the name alone, then the function has a good name.

Example

Good

- convertCelsuisToFahrenheit
- isValidLaunchCode
- updateMissionControl

Bad

- convertC

- validCode
 - msgToMC
-

If you find yourself writing a comment to describe what your function does, consider whether a better name might remove the need for such additional explanation. The best function (and variable) names are those that are *self-documenting*—descriptive enough not to need further explanation.

Using self-documenting names means that the code that *uses* your function will be more readable, since your explanatory comments are not visible where the function is used. Additionally, it is easy for comments to become inaccurate; when you update your code to change behavior, there is nothing forcing you to also update your comments. For this reason, some programmers live by the maxim, “Comments lie.” While we won’t go so far as to say that you should never use comments in your code, we *do* believe that comments should not be used to make up for poor function and variable names.

10.7.4 Check Your Understanding

Question

Which is the best name for the following function?

```
1 function
2   let
3     return
4
```

1. area
 2. calculateAreaOfCircle
 3. circle
 4. shape
-

10.8 Composing Functions

The practice of using functions to build other functions is known as **function composition**, or simply **composition**. To demonstrate, we consider a specific example.

10.8.1 Palindrome Checker

A palindrome can be defined as a word that is spelled the same backwards and forwards. Some examples are “tacocat”, “kayak”, and “racecar”.

Note

There are other factors that are sometimes included in the definition of a palindrome. For example, an alternative definition is that a palindrome is a sentence or phrase that contains letters in the same order, whether considered from beginning-to-end, or end-to beginning, ignoring punctuation, case, and spaces.

We want to write a boolean function—a function that returns `true` or `false`—to determine if a word is a palindrome.

One way to state the palindrome condition is to say that a palindrome is a string that is equal to its reverse. In other words, we can test for palindromes by taking a string, reversing it, and then comparing the reversed string to the original. If the two are equal, we have a palindrome.

To that end, it would be very useful to have a function that reverses a string, wouldn't it?

The `reverse` Function

Let's write a function that, given a string, returns its reverse.

One approach uses the accumulator pattern:

```
1  function
2    let      ''
3
4    for let    0
5
6
7    return
```

This is the same algorithm that we used previously to *reverse a string*.

Another approach is to use the fact that there is a `reverse` method for arrays, and that the methods `split` and `join` allow us to go from strings to arrays and back (this was covered in *Array Methods*).

```
1  function
2    let
3    let
4    return
```

[repl.it](#)

Let's break down the steps carried out by this function:

1. **Turn the string into an array of characters.** We call `str.split('')`, using the empty string as the splitting character, returns an array of the individual characters that make up the string.
2. **Reverse the array of characters.** To do this, we use the built-in array method `reverse`.
3. **Join the reversed character array into a string.** We call `.join('')`. Joining with the empty string is the same as concatenating each of the individual characters together into a single string.

Try It!

Use method chaining to reduce the `reverse` function to a single line. Open the link below the source code above to give it a shot.

The `isPalindrome` Function

Using our `reverse` function for strings, we can create our palindrome checker. Recall that our approach will be to take the string argument, reverse it, and then compare the reversed string to the original string.

```
1 function
2     return
3
4
5 function
6     return
7
```

repl.it

Since `isPalindrome` uses our `reverse` function, this is an example of composition.

Try It!

Does our `isPalindrome` function work? Run it yourself to see!

10.8.2 Functions Should Do Exactly One Thing

An important consideration when writing a function is size. By “size” we mean that functions should be short and, more importantly, *do exactly one thing*.

This principle is easier to state than to put into practice. For example, what if we had written `isPalindrome` without breaking out the `reverse` code into a separate function?

```
1 function
2     let
3         return
4
```

This function is still short, which is good. But does it do one thing (check if a string is a palindrome) or multiple things (check the string, *and* reverse a string)? This is a bit subjective, and here the answer is certainly debatable.

Some cases will be much more clear-cut, however. Consider the sandwich function, `makeSandwich`, from the section *A Good Function-Writing Process*. Suppose we wanted to expand the capability of our program to not only make a sandwich, but to also pour a beverage (to go along with our lunch). It would be a bad idea to amend our function to do both, ending up with a function that has a name like `makeSandwichAndPourDrink`.

A much better solution would look like this:

```
1 function /*parameters*/
2     // make the sandwich
3
4
5 function /*parameters*/
6     // pour the drink
7
8
9 function /*parameters*/
10    /*parameters*/
11    /*parameters*/
```

Why is this better? Smaller functions are easier to debug, for one thing. And by separating single responsibilities into individual functions, we also make our code easier to read and more reusable. In looking at the `makeLunch` function, it is very clear what is going on. First, it makes a sandwich, then it pours a drink.

Were the `makeLunch` function to simply contain all of the code necessary to carry out *both* tasks, there would be no clear separation between one task and the other, and the only way we might describe the various sections of the larger function would be to use comments. And, *as we've discussed*, comments should be a secondary option for explaining your code.

10.9 Why Create Functions?

After wading through all of the new syntax necessary to create a function, you might be asking yourself, *Why would I ever want to do this?* Good question! We have a few answers.

10.9.1 Functions Reduce Repetition

Like loops, functions help us keep our code DRY. When we need to repeat the same basic task in multiple parts of a program, a function will allow us to package up that task into a neat, reusable form. Loops enable the same task to be repeated many times in succession, while functions enable the same task to be repeated in different portions of a program.

10.9.2 Functions Make Your Code More Readable

Placing a piece of functionality within a function allows us to put a name on that functionality. Consider our *palindrome example*. One way to write that function is:

```
1 function
2
3   let      ''
4
5   for let    0
6
7
8   return
```

While the variable name `reversed` is descriptive, giving us a sense of what is going on with the `for` loop, the function becomes even more readable when we break out the reversing behavior into a separate function.

```
1 function
2   let      ''
3
4   for let    0
5
6
7   return
8
9
10 function
11   return
```

Aside from following the principle that functions should do only one thing, the logic within `isPalindrome` is more clear and self-descriptive. The function itself says, *a string is a palindrome if it is equal to its reverse*. To draw this same conclusion from the example above, without a `reverse` function, we are required to analyze more of the program's logic.

10.9.3 Functions Reduce Complexity

Large programs can be broken down into smaller parts using functions. Imagine a car built out of a single, large piece of metal. Were such a car to break down, diagnosing the problem would be difficult, and fixing it nearly impossible. The mechanic would have to determine where the issue was, then cut apart the bad portion, create a custom-made replacement portion, and weld it into place.

The complexity of this situation becomes much less when a car is made up of lots of small parts, each of which can be tested and replaced individually. The same thing happens with code. While there are many other organizational units for programs—including modules, files, and packages—functions are the most basic and universal organizational tool.

10.9.4 Functions Enable Code Sharing

Encapsulating behavior within a function makes it easy to reuse that code within a program, but it also allows you to share that behavior across files and even different projects. This becomes critically important when you start working on bigger programs that consist of a large number of files.

You will explore this idea in the *Modules chapter*.

10.9.5 Functions Save Millions of Lives Every Day

Okay, not really. But the point is, functions are incredibly powerful tools that you will come to appreciate and find indispensable. Ask a professional programmer if they could do their job without functions, and the answer will be an emphatic “NO!”

While functions may seem abstract and difficult to learn at first, repeated practice will lead to mastery. We promise that your work will be worth it.

10.10 Exercises: Functions

To become good at solving problems with code you need to be able to break large problems into small ones. Usually, these smaller problems will take the form of functions that are used to solve the larger problem. Therefore, to become good at solving problems you need to become good at writing functions. And to master functions, you need to write a *lot* of them.

These exercises ask you to write lots of relatively small functions, which combine to form larger, more complicated ones.

At the end, you will be able to create strings of shapes, like this nifty diamond:

```
#  
###  
####  
#####  
######  
######  
######  
####  
###  
#
```

There is no starter code for these exercises, so create a new Node.js project at [repl.it](#) to get started.

10.10.1 Rectangles

1. Write a function `line(size)` that returns a line with exactly `size` hashes.

5

Console Output

```
#####
```

2. Write a function `square(size)` that returns an `size` by `size` square of hashes. Use your `line` function to do this.

5

Console Output

```
#####
#####
```

Tip

The newline character, `\n`, will be helpful to you.

Warning

For this and all other functions in these exercises, make sure you do NOT have a newline character at the end of your string. Not only will `console.log` add a newline there for you, but having an extra newline at the end will make life harder for you toward the end of the exercises.

3. Write a function `rectangle(width, height)` that returns a rectangle with the given width and height. Use your `line` function to do this.

5 3

Console Output

```
#####
#####
```

4. Now, go back and rewrite `square` to use `rectangle`.

10.10.2 Triangles

1. Write a function `stairs(height)` that prints the staircase pattern shown below, with the given height. Use your `line` function to do this.

5

Console Output

```
#  
##  
###  
###  
#####
```

2. Write a function `spaceLine(numSpaces, numChars)` that returns a line with exactly the specified number of spaces, followed by the specified number of hashes, followed again by `numSpaces` more spaces.

```
3 5
```

Console Output

```
____#####____
```

Note

We have inserted underscores to represent spaces, so they are visible in the output. Don't do this in your code.

3. Write a function `triangle(height)` that returns a triangle of the given height.

```
5
```

Console Output

```
#  
##  
###  
###  
#####
```

Tip

Consider the top line of the triangle to be level 0, the next to be line 1, and so on. Then line `i` is a space-line with height `- i - 1` spaces and `2 * i + 1` hashes.

10.10.3 Diamonds

1. Write a function `diamond(height)` that returns a diamond where the *top* portion has the given height.

```
5
```

Console Output

```
#  
##  
###  
###  
#####  
#####  
#####  
#####
```

(continues on next page)

(continued from previous page)

```
#####  
###  
#
```

Tip

Consider what happens if you create a triangle and reverse it using *our reverse function*.

10.10.4 Bonus Mission

Refactor your functions so that they take a single character as a parameter, and draw the shapes with that character instead of always using '#'. Make the new parameter optional, with default value '#'.

10.11 Studio: Functions

The `reverse` method flips the order of the elements within an array. However, `reverse` does not affect the digits or characters within those elements.

Example

```
1 let      'hello'  'world'  123  'orange'  
2  
3  
4
```

Console Output

```
[ 'orange', 123, 'world', 'hello' ]
```

What if we wanted the reversed array to be `['egnaro', 321, 'dlrow', 'olleh']`?

Let's have some fun by creating a process that reverses BOTH the order of the entries in an array AND the order of characters within the individual elements.

Remember that a function should perform only one task. To follow this best practice, we will solve the array reversal by defining two functions - one that reverses the characters in a string (or the digits in a number) and one that flips the order of entries in the array.

10.11.1 Reverse Characters

1. In the *composing functions* section, we examined a function that *reverses the characters in a string* using the `split` and `join` methods. Let's rebuild that function now.
 - a. Define the function as `reverseCharacters`. Give it one parameter, which will be the string to reverse.
 - b. Within the function, `split` the string into an array, then reverse the array.
 - c. Use `join` to create the reversed string and *return* that string from the function.
 - d. Below the function, define and initialize a variable to hold a string.

- e. Use `console.log(reverseCharacters(myVariableName));` to call the function and verify that it correctly reverses the characters in the string.
- f. *Optional:* Use method chaining to reduce the lines of code within the function.

Code exercises 1 - 3 at [repl.it](#)

Tip

Use these sample strings for testing:

- a. 'apple'
 - b. 'LC101'
 - c. 'Capitalized Letters'
 - d. 'I love the smell of code in the morning.'
-

10.11.2 Reverse Digits

2. The `reverseCharacters` function works great on strings, but what if the argument passed to the function is a number? Using `console.log(reverseCharacters(1234));` results in an error, since `split` only works on strings (TRY IT). When passed a number, we want the function to return a number with all the digits reversed (e.g. 1234 converts to 4321 and NOT the string "4321").
 - a. Add an `if` statement to `reverseCharacters` to check the `typeof` the parameter.
 - b. If `typeof` is 'string', return the reversed string as before.
 - c. If `typeof` is 'number', convert the parameter to a string, reverse the characters, then convert it back into a number.
 - d. Return the reversed number.
 - e. Be sure to print the result returned by the function to verify that your code works for *both strings and numbers*. Do this before moving on to the next exercise.

Tip

Use these samples for testing:

- a. 1234
 - b. 'LC101'
 - c. 8675309
 - d. 'radar'
-

10.11.3 Complete Reversal

3. Now we are ready to finish our complete reversal process. Create a new function with one parameter, which is the array we want to change. The function should:
 - a. Define and initialize an empty array.
 - b. Loop through the old array.

- c. For each element in the old array, call `reverseCharacters` to flip the characters or digits.
 - d. Add the reversed string (or number) to the array defined in part ‘a’.
 - e. Return the final, reversed array.
- f. *Be sure to print the results from each test case in order to verify your code.*

Tip

Use this sample data for testing.

Input	Output
<code>['apple', 'potato', 'Capitalized Words']</code>	<code>['sdrow dezilatipaC', 'otatop', 'elppa']</code>
<code>[123, 8897, 42, 1138, 8675309]</code>	<code>[9035768, 8311, 24, 7988, 321]</code>
<code>['hello', 'world', 123, 'orange']</code>	<code>['egnaro', 321, 'dlrow', 'olleh']</code>

10.11.4 Bonus Missions

4. Define a function with one parameter, which will be a string. The function must do the following:
 - a. Have a clear, descriptive name.
 - b. Return only the last character from strings with lengths of 3 or less.
 - c. Return only the first 3 characters from strings with lengths larger than 3.
 - d. [Build your function at repl.it.](#)
5. Now test your function:
 - e. Outside of the function, define the variable `str` and initialize it with a string (e.g. `'Functions rock!'`).
 - f. Define a second variable and initialize it with `myFunctionName(str)`.
 - g. Use a template literal to print, We put the ‘ ’ in ‘ ’. Fill in the blanks with the values from `someNameThatIChose` and `str`.
6. The area of a rectangle is equal to its *length x width*.
 - a. Define a function and the required parameters to calculate the area of a rectangle.
 - b. The function should *return* the area, NOT print it.
 - c. Call your area function by passing in two arguments - the length and width.
 - d. Use a template literal to print, “The area is cm²”
 - e. *Optional:* If only one argument is passed to the function, then the shape is a square. Modify your code to deal with this case.
 - f. [Code the area function at repl.it.](#)

Tip

Use these test cases.

- a. `length = 2, width = 4 (area = 8)`

- b. length = 14, width = 7 (area = 98)
 - c. length = 20 (area = 400)
-

MORE ON FUNCTIONS

11.1 Functions as Values

Functions are powerful tools in any programming language, and JavaScript uses these tools in some flexible and creative ways. This chapter introduces a bit more of the power of functions.

11.1.1 Functions Are Data

We *defined* a value as “a specific piece of data.” Some examples are the number 42, the string "LC101", and the array ["MO", "FL", "DC"]. *Functions are also values*, and while they appear to be very different from other values we have worked with, they share many core characteristics.

In particular, functions have a data type, just like all other values. Recall that a **data type** is a group of values that share characteristics, such as strings and numbers. Strings share the characteristic of having a length, while numbers don’t. Numbers can be manipulated in ways that strings cannot, via operations like division and subtraction.

Example

The data type of the type conversion function Number is function. In fact, all functions are of type function.

```
1     typeof 42
2     typeof "LC101"
3     typeof
```

Console Output

```
number
string
function
```

Like other data types, functions may be assigned to variables. If we create a function named hello we can assign it to a variable with this syntax:

```
1 function
2   // function body
3
4
5
6
7 let
```

When a variable refers to a function, we can use the variable name to *call* the function:



The variable `helloFunc` can be thought of as an *alias* for the function `hello`. When we call the function `helloFunc`, JavaScript sees that it refers to the function `hello` and calls that *specific* function.

When we use a variable *name*, we are really using its *value*. If the variable `class` is assigned the value "LC101", then `console.log(class)` prints "LC101". When a variable holds a function, it behaves the same way as when it holds a number or a string. The variable *refers to* the function.

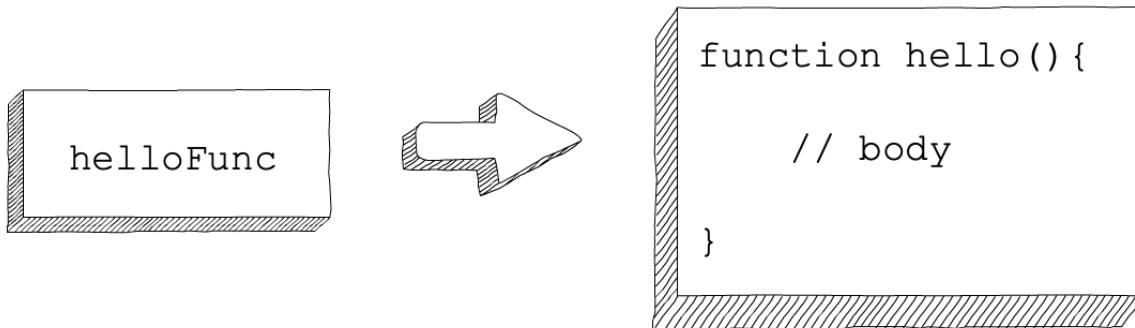


Fig. 1: A variable that refers to a function.

Again, *functions are values*. They can be used just like general values. For example:

- Functions may be assigned to variables.
- Functions may be used in expressions, such as comparisons.
- Functions may be converted to other data types.
- Functions may be printed using `console.log`.
- Functions may be passed as arguments to other functions.
- Functions may be returned from other functions.

Some of these function behaviors do not prove to be useful. You will probably never need to convert a function to a boolean, or ask whether a function is greater than 5. However, other behaviors, like passing functions as arguments and assigning them to variables, turn out to be *extremely* useful.

11.2 Anonymous Functions

You already know *one method for creating a function*:

```
1 function
2
3 // function body
4
5
```

A function defined in this way is a **named function** (`myFunction`, in the example above).

Many programming languages, including JavaScript, allow us to create **anonymous functions**, which *do not* have names. We can create an anonymous function by simply leaving off the function name when defining it:

```

1  function
2
3      // function body
4
5

```

You might be asking yourself, *How do I call a function if it doesn't have a name?!* Good question. Let's address that now.

11.2.1 Anonymous Function Variables

Anonymous functions are often assigned to variables when they are created, which allows them to be called using the variable's name.

Example

Let's create and use a simple anonymous function that returns the sum of two numbers.

```

1 let      function
2     return
3
4         1  1
5

```

Console Output

```

2

```

The variable `add` refers to the anonymous function created on lines 1 through 3. We call the function using the *variable* name, since the function doesn't have a name.

The visual analogy here is the same as that of a variable referring to a named function.

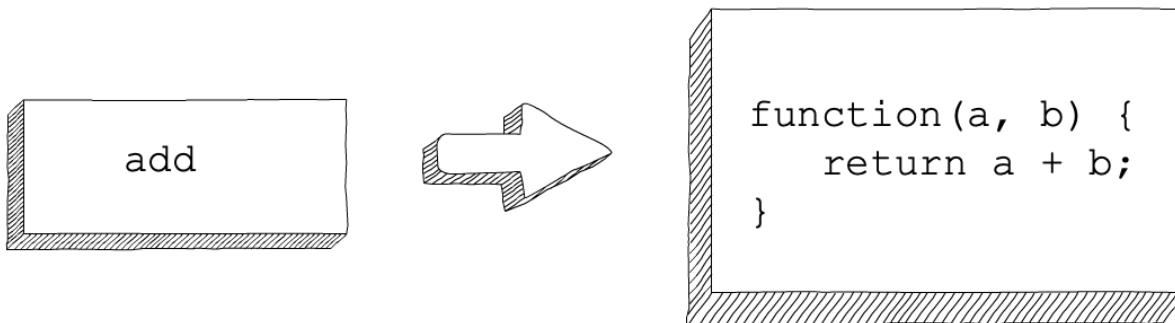


Fig. 2: A variable that refers to an anonymous function.

Warning

Like other variable declarations, an assignment statement using an anonymous function should be terminated by a semi-colon, ; . This is easy to overlook, since named functions do *not* end with a semi-colon.

11.2.2 Check Your Understanding

Question

Convert the following named function to an anonymous function that is stored in a variable.

```
1 function
2   let           "
3   let           "
4   return         "
```

repl.it

Question

Consider the code sample below, which declares an anonymous function beginning on line 1.

```
1 let      function
2   return
3
4
5 let
```

Which of the following are valid ways of invoking the anonymous function with the argument "abcd"? (Choose all that apply.)

1. f1("abcd");
 2. function("abcd");
 3. f2("abcd");
 4. It is not possible to invoke the anonymous function, since it doesn't have a name.
-

Question

Complete the following code snippet so that it logs an error message if userInput is negative.

```
1 let      function
2   "ERROR: "
3
4
5 if       0
6   "Invalid input"
7
```

repl.it

11.3 Passing Functions as Arguments

Functions are data, and therefore can be passed around just like other values. This means a function can be *passed to another function* as an argument. This allows the *function being called* to use the *function argument* to carry out its action. This turns out to be extremely useful.

Examples best illustrate this technique, so let's look at a couple now.

11.3.1 Example: setTimeout

The built-in function `setTimeout` allows a programmer to pass a function, specifying that it should be called at later point in time. Its basic syntax is:

```
setTimeout(func, delayInMilliseconds);
```

Example

Suppose we want to log a message with a 5 second delay. Since five seconds is 5000 milliseconds (1 second = 1000 milliseconds), we can do so like this:

```
1 function
2     "The future is now!"
3
4
5     5000
```

repl.it

Console Output

```
"The future is now!"
```

Try It!

Is the call to `printMessage` actually delayed? Don't just take our word for it, try this yourself. Play with our example to change the delay.

The function `printMessage` is *passed* to `setTimeout` the same as any other argument.

A common twist often used by JavaScript programmers is to use an *anonymous* function as an argument.

Example

This program has the same behavior as the one above. Instead of creating a named function and passing it to `setTimeout`, it creates an anonymous function within `setTimeout`'s argument list.

```
1     function
2         "The future is now!"
3
4     5000
```

Examples like this look odd at first. However, they become easier to read over time. Additionally, code that passes anonymous functions is ubiquitous in JavaScript.

11.3.2 Example: The Array Method `map`

The array method `map` allows for every element in an array to be *mapped* or *translated*, using a given function. Here's how to use it:

```
let mappedArray = someArray.map(func);
```

The argument `func` should take a single value from the array and return a new value. The returned array, `mappedArray`, contains each of the individual return values from the mapping function, `func`.

Example

```
1 let      3.14  42  4811
2
3 let          function
4   return    2
5
6
7 let
8
9
10
```

Console Output

```
[3.14, 42, 4811]
[ 6.28, 84, 9622 ]
```

Notice that `map` does *not* alter the original array.

When using `map`, many programmers will define the mapping function anonymously in the same statement as the method call `map`.

Example

This program has the same output as the one immediately above. The mapping function is defined anonymously within the call to `map`.

```
1 let      3.14  42  4811
2
3 let          function
4   return    2
5
6
7
```

Console Output

```
[ 6.28, 84, 9622 ]
```

11.3.3 Check Your Understanding

Question

Similar to the map example above, finish the program below to halve each number in an array.

```
1 let      3.14  42  4811
2
3 // TODO: Write a mapping function
4 // and pass it to .map()
5 let
6
7
```

repl.it

Question

Use the map method to map an array of strings. For each name in the array, map it to the first initial.

```
1 let      "Chris"  "Jim"  "Sally"  "Blake"  "Paul"
2
3 // TODO: Write a mapping function
4 // and pass it to .map()
5 let
6
7
```

repl.it

11.4 Receiving Function Arguments

The previous section illustrates how a function can be passed to another function as an argument. This section takes the opposite perspective to *write* functions that can take other functions as arguments.

11.4.1 Example: A Generic Input Validator

Our first example will be a generic input validator. It will prompt a user for input, using a parameter to the function to do the actual work of validating the input.

Example

```
1 const      'readline-sync'
2
3 function
4
5 // Prompt the user, using the prompt string that was passed
6 let
7
8 // Call the boolean function isValid to check the input
```

(continues on next page)

(continued from previous page)

```
9   while
10      "Invalid input. Try again."
11
12
13
14 return
15
16
17 // A boolean function for validating input
18 let      function
19   return 2    0
20
21
22      'Enter an even number:'
```

Sample Output

```
Enter an even number: 3
Invalid input. Try again.
Enter an even number: 5
Invalid input. Try again.
Enter an even number: 4
4
```

The function `getValidInput` handles the work of interacting with the user, while allowing the validation logic to be customized. This separates the different concerns of validation and user interaction, sticking to the idea that *a function should do only one thing*. It also enables more reusable code. If we need to get different input from the user, we can simply call `getValidInput` with different arguments.

Example

This example uses the same `getValidInput` function defined above with a different prompt and validator function. In this case, we check that a potential password has at least 8 characters.

```
1 const      'readline-sync'
2
3 function
4
5   let
6
7   while
8      "Invalid input. Try again."
9
10
11
12 return
13
14
15 let      function
16
17 // Passwords should have at least 8 characters
18 if      8
19   return false
20
21
```

(continues on next page)

(continued from previous page)

```
22     return true  
23  
24  
25     'Create a password: '
```

Sample Output

```
Create a password: launch  
Invalid input. Try again.  
Create a password: code  
Invalid input. Try again.  
Create a password: launchcode  
launchcode
```

Try It!

1. Use our `getValidInput` function to ensure user input starts with “a”.
2. Create another validator that ensures user input is a vowel.

[Try it at repl.it](#)

11.4.2 Example: A Logger

Another common example of a function using another function to customize its behavior is that of logging. Real-world applications are capable of logging messages such as errors, warnings, and statuses. Such applications allow for log messages to be sent to one or more destinations. For example, the application may log messages to both the console and to a file.

We can write a logging function that relies on a function parameter to determine the logging destination.

A Simple Logger

Example

The `logError` function outputs a standardized error message to a location determined by the parameter `logger`.

```
1 let          function  
2  
3   // Put the message in a file  
4  
5  
6  
7 function  
8   let          'ERROR: '  
9  
10  
11  'Something broke!'
```

Let’s examine this example in more detail.

There are three main program components:

1. Lines 1-5 define `fileLogger`, which takes a string argument, `msg`. We have not discussed writing to a file, but Node.js is capable of doing so.
2. Lines 7-10 define `logError`. The first parameter is the message to be logged. The second parameter is the logging function that will do the work of sending the message somewhere. `logError` doesn't know the details of how the message will be logged. It simply formats the message, and calls `logger`.
3. Line 12 logs an error using the `fileLogger`.

This is the flow of execution:

1. `logError` is called, with a message and the logging function `fileLogger` passed as arguments.
2. `logError` runs, passing the constructed message to `logger`, which refers to `fileLogger`.
3. `fileLogger` executes, sending the message to a file.

A More Complex Logger

This example can be made even more powerful by enabling multiple loggers.

Example

The call to `logError` will log the message to both the console and a file.

```
1 let          function
2
3   // Put the message in a file
4
5
6
7 let          function
8
9
10
11
12
13 function
14
15 let          'ERROR: '
16
17 for let      0
18
19
20
21
22   'Something broke!'
```

The main change to the program is that `logError` now accepts an *array* of functions. It loops through the array, calling each logger with the message string.

As with the validation example, these programs separate behaviors in a way that makes the code more flexible. To add or remove a logging destination, we can simply change the way that we call `logError`. The code *inside* `logError` doesn't know how each logging function does its job. It is concerned only with creating the message string and passing it to the logger(s).

11.4.3 A Word of Caution

What happens if a function expects an argument to be a function, but it isn't?

Try It!

```
1 function
2
3
4     "Al"
5
```

repl.it

Question

What type of error occurs when attempting to use a value that is NOT a function as if it were one?

11.5 Why Use Anonymous Functions?

At this point, you may be asking yourself *Why am I learning anonymous functions?* They seem strange, and their utility may not be immediately obvious. While the opinions of programmers differ, there are two main reasons why we think anonymous functions are important to understand.

11.5.1 Anonymous Functions Can Be Single-Use

There are many situations in which you will need to create a function that will only be used once. To see this, recall one of our earlier examples.

Example

The anonymous function created in this example cannot be used outside of `setTimeout`.

```
1 function
2     "The future is now!"
3 5000
```

Defining an anonymous function at the same time it is passed as an argument prevents it from being used elsewhere in the program.

Additionally, in programs that use lots of functions—such as web applications, as you will soon learn—defining functions anonymously, and directly within a function call, can reduce the number of names you need to create.

11.5.2 Anonymous Functions Are Ubiquitous in JavaScript

JavaScript programmers use anonymous functions *a lot*. Many programmers use anonymous functions with the same gusto as that friend of yours who puts hot sauce on *everything*.

Just because an anonymous function isn't needed to solve a problem doesn't mean that it *shouldn't* be used to solve the problem. Avoiding JavaScript code that uses anonymous functions is impossible.

Any programming problem in JavaScript can be solved *without* using anonymous functions. Thus, the extent to which you use them in your own code is somewhat a matter of taste. We will take the middle road throughout the rest of this course, regularly using both anonymous and named functions.

11.5.3 Check Your Understanding

Question

Explain the difference between named and anonymous functions, including an example of how an anonymous function can be used.

11.6 Recursion

11.6.1 Quick Review

In the previous chapter, we learned how to define a function and its parameters.

Example

```
function
  return      2
              12
```

Console Output

```
14
```

When called, the parameter num is passed an argument, which in this case is the number 12. The function executes and returns the value 14, which the `console.log` statement prints.

Functions Can Call Other Functions

Functions should only accomplish one (preferably simple) task. To solve more complicated tasks, one small function must call other functions.

Example

```
function
  return      2
              12
              let      3
```

(continues on next page)

(continued from previous page)

```
return
```

12

Console Output

```
17
```

Of course, there is no need to write a function to add 5 to a value, but the example demonstrates calling a function from within another function.

11.6.2 What Is Recursion?

In programming, the *divide and conquer* strategy solves a problem by breaking it down into smaller, simpler pieces. If these pieces *can all be solved in exactly the same way*, then we gain an additional advantage. Solving the big problem becomes a process of completing and combining the smaller parts.

Splitting up a large task into smaller, identical pieces allows us to reuse a single function rather than coding several different functions. We accomplish this by either:

- a. Setting up a loop to call one function lots of times, OR
- b. Building a function that splits up the large problem for us, until a *simplest case* is found and solved.

Recursion is the process of solving a larger problem by breaking it into smaller pieces that *can all be solved in exactly the same way*. The clever idea behind recursion is that instead of using a loop, a function simply calls *itself* over and over again, with each step reducing the size of the problem.

Through recursion, a problem eventually gets reduced to a very simple task, which can be immediately solved. This small answer sets up the solution for the previous step, which in turn solves the next bigger step. Properly built, the function combines all of the small answers to solve the original problem.

Many new programmers (and even veteran ones) find recursion an abstract and tricky concept. One helpful way to approach the idea is to walk through an example.

11.7 Recursion Walkthrough: The Base Case

To ease into the concept of recursion, let's start with a loop task.

In the Arrays chapter, we examined the *join method*, which combines the elements of an array into a single string. If we have `arr = ['L', 'C', '1', '0', '1']`, then `arr.join('')` returns the string 'LC101'.

We can reproduce this action with either a `for` or a `while` loop.

11.7.1 Joining Array Elements With a Loop

Examine the code samples below:

<p>Use a <code>for</code> loop to iterate through the array and add each entry into the <code>newString</code> variable.</p>	<p>Use a <code>while</code> loop to add the first element in the array to <code>newString</code>, then remove that element from the array.</p>
<pre> 1 let 'L' 'C' '1' '0' '1' 2 let '' 3 4 for 0 5 6 7 8 9 Console Output 'LC101' ['L', 'C', '1', '0', '1'] </pre>	<pre> 1 let 'L' 'C' '1' '0' '1' 2 let '' 3 4 while 0 5 6 7 8 9 Console Output 'LC101' [] </pre>

Inside each loop, the code simply adds two strings together—whatever is stored in `newString` plus one element from the array. In the `for` loop, the element is the next item in the sequence of entries. In the `while` loop, the element is always the first entry from whatever remains in the array.

OK, the loops join the array elements together. Now let's see how to accomplish the same task without a `for` or `while` statement.

11.7.2 Bring In Recursion Concepts

First, state the problem to solve: *Combine the elements from an array into a string.*

Second, split the problem into small, identical steps: Looking at the loops above, the “identical step” is just adding two strings together - `newString` and the next entry in the array.

Third, build a function to accomplish the small steps: Let's call the function `combineEntries`, and we will set an array as the parameter.

```

function
//TODO: Add code here

```

We want `combineEntries` to repeat over and over again until the task is complete.

How do we make this happen without using `for` or `while`?

11.7.3 Identifying the Base Case

`for` and `while` loops end when a particular condition evaluates to `false`. In the examples above, these conditions are `i < arr.length` and `arr.length > 0`, respectively.

With recursion, we do not know how many times `combineEntries` must be called. To make sure the code stops at the proper time, we need to identify a condition that ends the process. This is called the **base case**, and it represents the simplest possible task for our function.

If the base case is `true`, the recursion ends and the task is complete. If the base case is `false`, the function calls itself again.

We check for the base case like this:

```
1 function
2   if           true
3     //solve last small step
4     //end recursion
5   else
6     //call combineEntries again
7
8
```

For the joining task, the *base case* occurs when we pass in a one-element array (e.g. `['L']`). With no other elements to join together, the function just needs to return `'L'`.

Let's update `combineEntries` to check if the array contains only one item.

```
1 function
2   if           1
3     return      0
4   else
5     //call combineEntries again
6
7
```

`arrayName.length <= 1` sets up the condition for ending the recursion process. If it is `true`, the single entry gets returned, and the function stops. Otherwise, `combineEntries` gets called again.

Note

We define our base case as `arrayName.length <= 1` rather than `arrayName.length === 1` just in case an empty array `[]` gets passed to the function.

11.7.4 The Case for the Base

What if we accidentally typed `arrayName.length === 2` as the condition for ending the recursion? If so, it evaluates to `true` for the array `['0', '1']`, and the function returns `'0'`. However, this leaves the element `'1'` in the array instead of adding it to the string. By mistyping the condition, we ended the recursion process too soon.

Similarly, if we used `arrayName[0] === 'Rutabaga'` as the condition, then any array that does NOT contain the string `'Rutabaga'` would never match the base case. In situations where the base case cannot be reached, the recursion process either throws an error, or it continues without end—an infinite loop.

Correctly identifying and checking for the base case is *critical* to building a working recursive process.

11.7.5 Check Your Understanding

Question

We can use recursion to remove all of the `'i'` entries from the array `['One', 'i', 'c', 'x', 'i', 'i', 54]`.

Consider the code sample below, which declares the `removeI` function.

```
1 function
2     if           true
3         //return final array
4         //end recursion
5     else
6         //remove one 'i' entry from array
7         //call removeI function again
8
9
```

Which TWO of the following work as a base case for the function? Feel free to test the options in the repl.it to check your thinking.

1. `!arr.includes('i')`
2. `arr.includes('i')`
3. `arr.indexOf('i') === -1`
4. `arr.indexOf('i') !== -1`

Experiment with this [repl.it](#).

Question

The **factorial** of a number ($n!$) is the product of a positive, whole number and all the positive integers below it.

For example, four factorial is $4! = 4 * 3 * 2 * 1 = 24$, and $5! = 5 * 4 * 3 * 2 * 1 = 120$.

Consider the code sample below, which declares the `factorial` function.

```
1 function
2     if           true
3         //solve last step
4         //end recursion
5     else
6         //call factorial function again
7
8
```

Which of the following should be used as base case for the function?

1. `integer === 1`
2. `integer < 1`
3. `integer === 0`
4. `integer < 0`

Experiment with this [repl.it](#).

11.8 Making A Function Call Itself

Congratulations! Identifying the base case is often the trickiest part of building a recursive function.

We've made it this far with `combineEntries`:

```

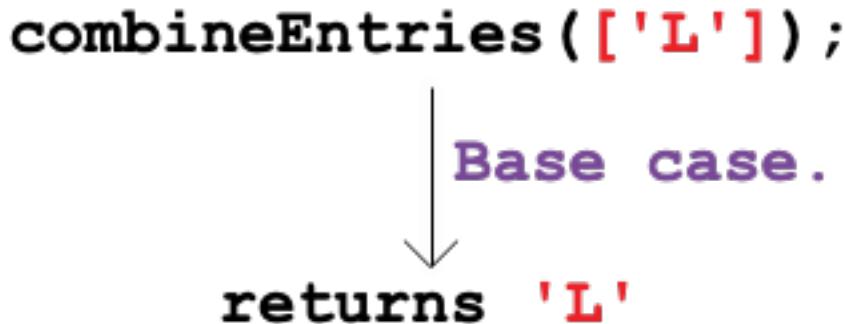
1 function
2   if           1
3     return      0
4   else
5     //call combineEntries again
6
7

```

Now we are ready to take the next step.

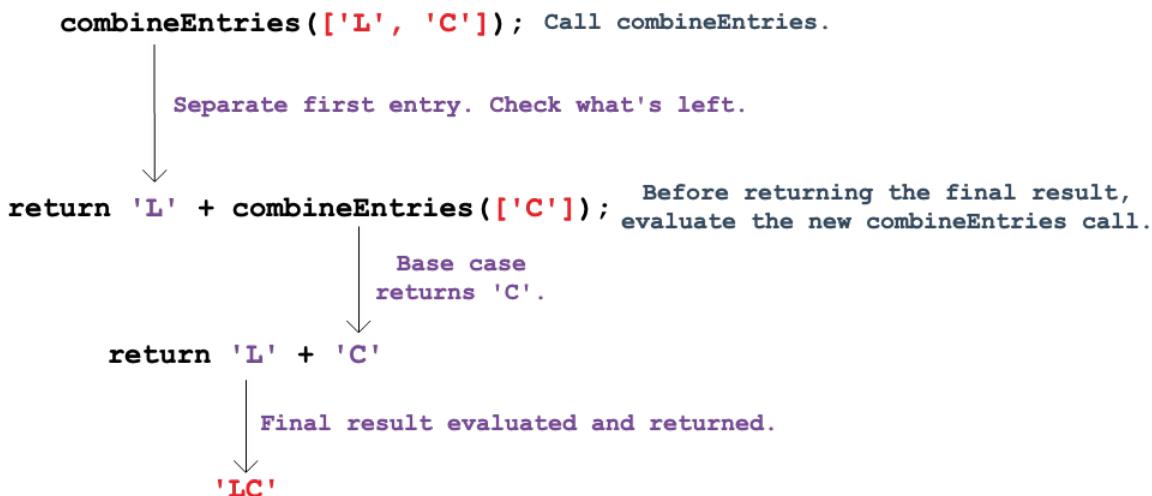
11.8.1 A Visual Representation

To help visualize what happens during recursion, let's start with the base case `['L']`:



Nothing complicated here. `combineEntries` sees only one item in the array, so it returns `'L'`.

Now consider an array with two elements, `['L', 'C']`:



In this case, `combineEntries` executes the `else` statement. We have no code for this yet, but we can still consider the logic:

- `combineEntries` returns `'L'` and calls itself again using what is left inside the array (`['C']`).

- b. When passed `['C']`, which is the base case, `combineEntries` returns `'C'`.
- c. The strings `'L'` and `'C'` get combined and returned as the final result.

Next, consider an array with three elements `['L', 'C', '1']`:

```
combineEntries(['L', 'C', '1']); Call combineEntries.
    ↓
    Separate first entry. Check what's left.

return 'L' + combineEntries(['C','1']); Evaluate the new combineEntries call.
    ↓
    Separate new first entry. Check what's left.

return 'L' + ('C' + combineEntries(['1'])); Evaluate new combineEntries call.
    ↓
    Base case returns '1'.

return 'L' + ('C' + '1');

    ↓
    Final result evaluated and returned.

'LC1'
```

As before, `combineEntries` executes the `else` statement, and we can follow the logic:

- a. `combineEntries` returns `'L'` and calls itself again using what is left inside the array (`['C', '1']`).
- b. When passed `['C', '1']`, `combineEntries` returns `'C'` and calls itself again using what is left inside the array (`['1']`).
- c. When passed `['1']`, which is the base case, `combineEntries` returns `'1'`.
- d. The strings `'C'` and `'1'` get combined and returned.
- e. The strings `'L'` and `'C1'` get combined and returned as the final result.

As we make the array longer, `combineEntries` calls itself more times. Each call evaluates a smaller and smaller section of the array until reaching the base case. This sets up a series of return events - each one selecting a single entry from the array. Rather than building `'LC101'` from left to right, recursion constructs the string starting with the base case and adding new characters to the front:

Value Returned	Description
<code>'1'</code>	Base case. Returns the element from an array of length 1.
<code>'01'</code>	Combines the first element from an array of length 2 with the base case value.
<code>'101'</code>	Combines the first element from an array of length 3 with the two previous values.
<code>'C101'</code>	Combines the first element from an array of length 4 with the three previous values.
<code>'LC101'</code>	Combines the first element from an array of length 5 with the four previous values.

Recursive processes all follow this approach. Each call to the function reduces a problem into a slightly smaller piece. The reduction continues until reaching the simplest possible form—the base case. The base case is then solved, and this creates a starting point for completing all of the previous steps.

11.8.2 A Function Calls Itself

So how do we code the `else` statement in `combineEntries`? Recall what needs to happen each time the statement runs:

- a. Select the first element in the array,
- b. Call `combineEntries` again with a smaller array.

Bracket notation takes care of part a: `arrayName[0]`.

For part b, remember that the *slice method* returns selected entries from an array. To return everything BUT the first entry in `arr = ['L', 'C', '1', '0', '1']`, use `arr.slice(1)`.

Let's add the bracket notation and the `slice` method to our function:

```
1 function
2   if           1
3     return      0
4   else
5     return      0           1
6
7
```

Each time the `else` statement runs, it extracts the first element in the array with `arrayName[0]`, then it calls itself with the remaining array elements (`arrayName.slice(1)`).

For `combineEntries(['L', 'C', '1', '0', '1'])`, the sequence would be:

Step	Description
1	First call: Combine 'L' with <code>combineEntries(['C', '1', '0', '1'])</code> .
2	Second call: Combine 'C', with <code>combineEntries(['1', '0', '1'])</code> .
3	Third call: Combine '1', with <code>combineEntries(['0', '1'])</code> .
4	Fourth call: Combine '0', with <code>combineEntries(['1'])</code> .
5	Fifth call: Base case returns '1'.

To get the final result, proceed *up the chain*:

Step	Description
5	Return '1' to the fourth call.
4	Return '01' to the third call.
3	Return '101' to the second call.
2	Return 'C101' to the first call.
1	Return 'LC101' as the final result.

See this recursion in action.

11.8.3 Check Your Understanding

Question

What if we wanted to take a number (`n`) and add it to all of the positive integers below it? For example, if `n = 5`, the function returns $5 + 4 + 3 + 2 + 1 = 15$.

Consider the code sample below, which declares the `decreasingSum` function.

```
1 function
2     if           1
3         return
4     else
5         //call decreasingSum function again
6
7
```

Which of the following should be used in the `else` statement to recursively call `decreasingSum` and eventually return the correct answer?

1. `return integer + (integer-1);`
2. `return integer + (decreasingSum(integer));`
3. `return integer + (decreasingSum(integer-1));`
4. `return decreasingSum(integer-1);`

Experiment with this [repl.it](#).

11.9 Recursion Wrap-Up

In order to function (ba-dum chhhh), recursion must fulfill four conditions:

1. A series of small, identical steps combine to solve a larger problem.
2. A base case must be defined. When true, this simplest case halts the recursion.
3. A recursive function repeatedly calls itself.
4. Each time the recursive function is called, it must alter the data/variables/conditions in order to move closer to the base case.

Benefits of Recursion:

- a. Fewer lines of code required to accomplish a task,
- b. Makes code cleaner and more readable.

Drawbacks of Recursion:

- a. More abstract than using loops,
- b. Code is “more readable” only if the reader understands recursion.

11.9.1 Recursion in a Nutshell

- a. Build a single function to break a big problem into a slightly smaller version of the *exact same problem*.
- b. The function repeatedly calls itself to reduce the problem into smaller and smaller pieces.
- c. Eventually, the function reaches a simplest case (the *base*), which it solves.
- d. Solving the base case sets up the solutions to all of the previous steps.

11.9.2 Why Do I Need To Know Recursion?

If you ask veteran programmers how often they use recursion, you will get answers ranging from “Not since I had to do it in school,” to “Very regularly.” Some programmers avoid recursion like the plague, while others look forward to using it wherever it fits.

Most of the recursion problems you encounter in your tech career can be solved with loops instead. However, *recursion is a skill most programmers will see and are expected to know*, even if they do not use it all the time. How deep you need to dive depends entirely on the type of job you get, your team members, and your personal preference.

Let’s use an analogy. At some point in time, most teens must “solve a quadratic” in school (e.g. find ‘x’ in $x^2 + 2x - 35 = 0$). Perhaps you fondly remember doing this yourself. As kids, we were *expected* to know how to solve a quadratic, but as adults, the need to do this varies. Some of us must frequently find x, while others only need to solve one or two equations a year. Still others do not see quadratics again until their own kids learn about them.

Since their future jobs might not require it, why do teens need to learn how to solve quadratics? Because at some point in time they will have to do it again (if only to shock their kids), and they need to be ready when that happens.

The same is true for recursion.

Learn it. Love it. Use it.

11.10 Exercises: More Functions

11.10.1 Practice Yer Skills

Arrr! Welcome back, swabbie. Ye be almost ready fer yer next task—earning gold fer yer cap’n.

First, ye need to show what ye’ve learned.

Complete the Map

Not THAT kinda map, ye rat! Fold that up and do the following:

1. Create an anonymous function and set it equal to a variable. Yer function should:
 1. If passed a number, return the tripled value.
 2. If passed a string, return the string “ARRR!”
 3. If NOT passed a number or string, return the data unchanged.
 4. [Build yer function here](#), and be sure to test it.
2. Add to yer code! Use yer function and the *map method* to change the array `['Elocution', 21, 'Clean teeth', 100]` as follows:
 1. Triple all the numbers.
 2. Replace the strings with “ARRR!”
 3. Print the new array to confirm yer work.

11.10.2 Raid Yonder Shuttle

Shiver me timbers! Ye did well. Yer ready ta join a boarding party.

Ye may still be wonderin' *Why the blazes would I ever use an anonymous function?* Fer today's mission, o' course! We arrrr going to loot yonder shuttle, but since it's the 21st century, we need to do better than grappling hooks and rope.

Ye arrrr going to hack into the shuttle code and steal supplies. Since the LaunchCode TAs keep a sharp eye on the goodies, ye can't just add new functions like `siphonFuel` or `lootCargo`. Ye need to be more sneaky.

Clever.

Invisible.

Anonymous.

The first mate swiped a copy of the code ye need ta hack:

```
1  function
2    if      100000
3      return 'green'
4    else if      50000
5      return 'yellow'
6    else
7      return 'red'
8
9
10
11 function
12   if      7
13     return `Spaces available: ${7} `
14   else if      7
15     return `Over capacity by ${7} items.`
16   else
17     return "Full"
18
19
20
21 let      200000
22 let      'meal kits'  'space suits'  'first-aid kit'  'satellite'  'gold'
23   ↪'water'  'AE-35 unit'
24
25      "Fuel level: "
      "Hold status: "
```

Hack the code at repl.it.

1. First, steal some fuel from the shuttle:

1. Define an anonymous function and set it equal ter a variable with a normal, non-suspicious name. The function needs one parameter, which will be the fuel level on the shuttle.
 2. Ye must siphon off fuel without alerting the TAs. Inside yer function, ye want to reduce the fuel level as much as possible WITHOUT changing the color returned by the `checkFuel` function.
 3. Once ye figure out how much fuel ter pump out, return that value.
 4. Decide where to best place yer function call to gather our new fuel.
 5. Be sure to test yer function! Those bilge rat TAs will notice if they lose too much fuel.
2. Next, liberate some of that glorious cargo.
1. Define another anonymous function with an array as a parametarrrrr, and set it equal to another innocent variable.

2. Ye need to swipe two items from the cargo hold. Choose well. Stealing water ain't gonna get us rich. Put the swag into a new array and return it from the function.
 3. The cargo hold has better security than the fuel tanks. It counts how many things are in storage. Ye need to replace what ye steal with something worthless. The count MUST stay the same, or ye'll get caught and thrown into the LaunchCode brig.
 4. Don't get hasty, swabbie! Remember to test yer function.
3. Finally, ye need to print a receipt for the accountant. Don't laugh! That genius knows MATH and saves us more gold than ye can imagine.
1. Define a function that takes yer fuel and cargo functions as parametarrrrs.
 2. Use a template literal to print out, "Raided _____ kg of fuel from the tanks, and stole _____ and _____ from the cargo hold."

11.11 Studio: More Functions

11.11.1 Sort Numbers For Real

Recall that using the `sort` method on an array of numbers *produced an unexpected result*, since JavaScript converts the numbers to strings by default. Let's fix this!

Here is one approach to sorting an array:

1. Find the minimum value in an array,
2. Add that value to a new array,
3. Remove the entry from the old array,
4. Repeat steps 1 - 3 until the numbers are all in order.

Part A: Find the Minimum Value

Create a function with an array of numbers as its parameter. The function should iterate through the array and return the minimum value from the array.

Hint: Use what you know about `if` statements to identify and store the smallest value within the array.

Tip

Use this sample data for testing.

```
1  5  10  2  42
2  2  0   10  44  5  3  0   3
3  200 5   4   10  8  5   3.3  4.4  0
```

Code studio part A at repl.it.

Part B: Create a New Sorted Array

Create another function with an array of numbers as its parameter. Within this function:

- a. Define a new, empty array to hold the final sorted numbers.

- b. Use your function from the previous exercise to find the minimum value in the old array.
- c. Add the minimum value to the new array, and remove the minimum value from the old array.
- d. Repeat parts b & c until the old array is empty.
- e. Return the new sorted array.
- f. *Be sure to print the results in order to verify your code.*

Code studio part B at [repl.it](#).

Tip

Which type of loop?

Either a `for` or `while` loop will work inside this function, but one IS a better choice. Consider what the function must accomplish vs. the behavior of each type of loop. Which one best serves if the array has an unknown length?

11.11.2 More on Sorting Numbers

The sorting approach used above is an example of a *selection sort*. The function repeatedly checks an array for the minimum value, then places that value into a new container.

Selection sorting is NOT the most efficient way to accomplish the task, since it requires the function to pass through the array once for each item within the array. This takes way too much time for large arrays.

Fortunately, JavaScript has an elegant way to properly sort numbers.

Tip

Here is a nice, visual comparison of [different sorting methods](#).

Feel free to Google “bubble sort JavaScript” to explore a different way to order numbers in an array.

11.11.3 Part C: Number Sorting the Easy Way

If you Google “JavaScript sort array of numbers” (or something similar), many options appear, and they all give pretty much the same result. The sites just differ in how much detail they provide when explaining the solution.

One reference is here: [W3Schools](#).

End result: the JavaScript syntax for numerical sorting is `arrayName.sort(function(a, b) {return a-b});`.

Here, the anonymous function determines which element is larger and swaps the positions if necessary. This is all that `sort` needs to order the entire array.

Using the syntax listed above:

- a. Sort each sample array in increasing order.
- b. Sort each sample array in decreasing order.
- c. Does the function alter `arrayName`?
- d. Did your sorting function from part B alter `arrayName`?

Code studio part C at [repl.it](#).

11.11.4 So Why Write A Sorting Function?

Each programming language (Python, Java, C#, JavaScript, etc.) has built-in sorting methods, so why did we ask you to build one?

It's kind of a programming right of passage - design an efficient sorting function. Also, sorting can help you land a job.

As part of a tech interview, you will probably be asked to do some live-coding. One standard, go-to question is to sort an array WITHOUT relying on the built in methods. Knowing how to think though a sorting task, generate the code and then clearly explain your approach will significantly boost your appeal to an employer.

11.11.5 Bonus Mission

Refactor your sorting function from Part B to use recursion.

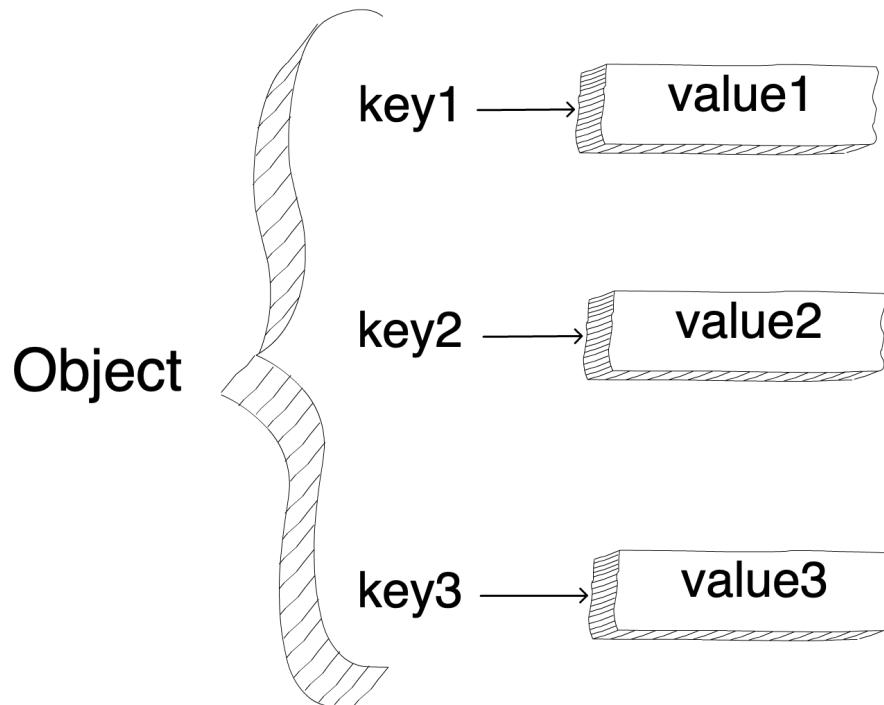
OBJECTS AND THE MATH OBJECT

12.1 Objects and Why They Matter

So far we have learned a lot about arrays, which are data structures that can hold many values. **Objects** are also data structures that can hold many values.

Unlike arrays, each value in an object has a name or **key** for reference purposes. The pairing between a key and its value is called a **key/value pair**.

Objects store as many key/value pairs as needed, and each value needs a key. Without a key, the value cannot be accessed or modified by the programmer.



12.1.1 Initializing Objects

When defining an object, we call the initialization an **object literal**. Objects require three things for the definition: a name, a set of keys, and their corresponding values.

Note

Object literals use curly braces, { }, to enclose the key/value pairs.

Once we have these three things, we write an object literal like so:

```
1 let
```

If we have a lot of key/value pairs in our object, we can also put each one on a separate line!

```
1 let  
2  
3  
4  
5  
6  
7  
8
```

Warning

When putting the key/value pairs on separate lines, it is important to pay attention to spaces and tabs! Incorrect spacing or tab usage can result in a bug.

When defining an object, keep in mind that the keys can only be valid JavaScript strings. The values can be any of the data types that we have previously discussed.

Example

Let's say that we want to create a small program for a zoo. We could create an object for storing the different data points about the animals in a zoo. We start with our first tortoise. His name is Pete! He is an 85 year old, 919 lb Galapagos Tortoise, who prefers a diet of veggies. Our object literal for all of this important data about Pete would be:

```
1 let  
2     "Galapagos Tortoise"  
3     "Pete"  
4     919  
5     85  
6     "pumpkins"  "lettuce"  "cabbage"
```

12.1.2 Methods and Properties

A **property** of an object is a key/value pair of an object. The property's name is the key and the property's value is the data assigned to that key.

A **method** performs an action on the object, because it is a property that stores a function.

Example

In the case of Pete, our zoo's friendly Galapagos Tortoise, the object `tortoiseOne` has several properties for his species, name, weight, age, and diet. If we wanted to add a method to our object, we might add a function that returns a helpful statement for the general public.

```
1 let
2     "Galapagos Tortoise"
3     "Pete"
4     919
5     85
6     "pumpkins"  "lettuce"  "cabbage"
7     function
8     return this      " is a "    this
9
10
```

In the example above, on line 8, we see a keyword which is new to us. Programmers use the `this` keyword when they call an object's property from within the object itself. We could use the object's name instead of `this`, but `this` is shorter and easier to read. For example, the method, `sign`, could have a return statement of `tortoiseOne.name + " is a " + tortoiseOne.species"`. However, that return statement is bulky and will get more difficult to read with more references to the `tortoiseOne` object.

12.1.3 Check Your Understanding

Question

Which of the following is NOT a true statement about objects?

- a. Objects can store many values
 - b. Objects have properties
 - c. Objects have methods
 - d. Keys are stored as numbers
-

Question

Which keyword can be used to refer to an object within an object?

- a. Object
 - b. let
 - c. this
-

12.2 Working with Objects

12.2.1 Accessing Properties

When using objects, programmers oftentimes want to retrieve or change the value of one of the properties. To access the value of a property, you will need the object's name and the key of the property.

Programmers have two ways to access the value of property:

1. Bracket syntax
2. Dot notation.

Bracket Syntax

To access a property with bracket syntax, the code looks like: `object ["key"]`.

Dot Notation

To access a property with dot notation, the code looks like: `object .key`. Notice that the key is no longer surrounded by quotes. However, keys are still strings.

Note

Recall, the only restraint in naming a key is that it has to be a valid JavaScript string. Since a key could potentially have a space in it, bracket syntax would be the only way to access the value in that property because of the quotes.

Example

```
1 let
2     "Galapagos Tortoise"
3     "Pete"
4     919
5     85
6     "pumpkins"  "lettuce"  "cabbage"
7
8
9     "name"
```

Console Output

```
Pete
Pete
```

12.2.2 Modifying Properties

A programmer can modify the value of a property by using either notation style.

Warning

Recall that mutability means that a data structure can be modified without making a copy of that structure. Objects are mutable data structures. When you change the value of a property, the original object is modified and a copy is NOT made.

Example

In our zoo software, we may want to update Pete's weight as he has gained 10 lbs. We will use both bracket syntax and dot notation for our software, but that is not a requirement! Feel free to use whichever one suits your needs and is easiest for you and your colleagues to read.

```
1 let
2     "Galapagos Tortoise"
3     "Pete"
4     919
5     85
6     "pumpkins"  "lettuce"  "cabbage"
7
8
9
10
11     10
12
13     "weight"
14
15     "weight"
```

Console Output

```
919
929
```

12.2.3 Check Your Understanding

All of the questions below refer to an object called `giraffe`.

```
1 let
2     "Reticulated Giraffe"
3     "Cynthia"
4     1500
5     15
6     "leaves"
7
```

Question

We want to add a method after the `diet` property for easily increasing Cynthia's age on her birthday. Which of the following is missing from our method? You can select MORE than one.

```
birthday: function () {age = age + 1;}
```

- a. return
- b. this
- c. diet

d. a comma

Question

Could we use bracket syntax, dot notation, or both to access the properties of `giraffe`?

12.3 Coding With Objects

12.3.1 Booleans and Objects

Objects are not stored by their properties or by value, but by *reference*. Storing something by reference means that it is stored based on its location in memory. This can lead to some confusion when comparing objects.

Example

Let's see how this affects our zoo software! Surely, the zoo has more than one tortoise. The second tortoise is named Patricia!

```
1 let
2     "Galapagos Tortoise"
3     "Patricia"
4     800
5     85
6     "pumpkins"  "lettuce"  "cabbage"
7     function
8     return this      " is a "    this
9
10
```

Because Pete and Patricia are members of the same species, are the same age, and have the same diet, you might notice that many of their properties are equal, but some are not. Pete weighs more than Patricia and of course, they have different names!

For this example, we will only keep the `species` and `diet` properties.

```
1 let
2     "Galapagos Tortoise"
3     "pumpkins"  "lettuce"  "cabbage"
4
5
6 let
7     "Galapagos Tortoise"
8     "pumpkins"  "lettuce"  "cabbage"
9
10
11
```

Console Output

```
false
```

The objects contain properties that have the same keys and equal values. However, the output is `false`.

Even though `tortoiseOne` and `tortoiseTwo` have the same keys and values, they are stored in separate locations in memory. This means that even though you can compare the properties in different objects for equality, you cannot compare the objects themselves for equality.

12.3.2 Iterating Through Objects

We can iterate through all of the values in an object, much like we would do with an array. We will use a `for` loop to do that, but with a slightly different structure. `for...in` loops are specifically designed to loop through the properties of an object. Each iteration of the loop accesses a key in the object. The loop stops once it has accessed every property.

Example

```
1 let
2   "Reticulated Giraffe"
3   "Cynthia"
4   1500
5   15
6   "leaves"
7
8
9 for     in
10      ","
11
```

Console Output

```
species, Reticulated Giraffe
name, Cynthia
weight, 1500
age, 15
diet, leaves
```

In this example, `item` is a variable that holds the string for each key. It is updated with each iteration of the loop.

Note

Inside a `for..in` loop, we can only use bracket syntax to access the property values.

Try It!

Write a `for..in` loop to print to the console the values in the `tortoiseOne` object. Try it at repl.it

12.3.3 Objects and Functions

Programmers can pass an object as the input to a function, or use an object as the return value of the function. Any change to the object within the function will change the object itself.

Example

```
1 let
2     "Reticulated Giraffe"
3     "Cynthia"
4     1500
5     15
6     "leaves"
7
8
9 function
10
11     return 1
12
13
14
15
16
17
18
```

Console Output

```
15
16
```

On line 16, when the `birthday` function is called, `giraffe` is passed in as an argument and returned. After the function call, `giraffe.age` increases by 1.

12.3.4 Check Your Understanding

Question

What type of loop is designed for iterating through the properties in an object?

Question

Given the following object definitions, which statement returns `true`?

```
1 let
2     150
3         "Galapagos Tortoise"
4         "pumpkins" "lettuce" "cabbage"
5
6
7 let
8     150
9         "Galapagos Tortoise"
10        "pumpkins" "lettuce" "cabbage"
11
```

- a. `tortoiseOne == tortoiseTwo`
- b. `tortoiseOne === tortoiseTwo`
- c. `tortoiseOne.age === tortoiseTwo.age`

12.4 The Math Object

JavaScript provides several built-in objects, which can be called directly by the user. One of these is the `Math` object, which contains more than the standard mathematical operations (`+`, `-`, `*`, `/`).

In the previous sections, we learned how to construct, modify, and use objects such as `giraffe`. However, JavaScript built-in objects cannot be modified by the user.

Unlike other objects, the `Math` object is *immutable*.

12.4.1 Math Properties Are Constants

The `Math` object has 8 defined properties. These represent *mathematical constants*, like the value for pi (π) or the square root of 2.

Instead of defining a variable to hold as many digits of pi as we can remember, JavaScript stores the value for us. To use this value, we do NOT need to create a new object. By using dot notation and calling `Math.PI`, we can access the value of pi.

Example

```
1
2
3
4
```

Console Output

```
3.141592653589793
12.566370614359172
6.283185307179586
```

As stated above, the properties within `Math` *cannot* be changed by the user.

Example

```
1
2
3
4
5
1234
```

Console Output

```
3.141592653589793
3.141592653589793
```

To use one of the other constants stored in `Math`, we replace `PI` with the property name (e.g. `SQRT2` stores the value for the square root of 2).

12.4.2 Other Math Properties

Besides the value of pi, JavaScript provides 7 other constants. How useful you find each of these depends on the type of project you need to complete.

More powerful uses of the Math object involve using *methods*, which we will examine next.

12.5 Math Methods

As with strings and arrays, JavaScript provides some built-in *methods* for the Math object. These allow us to perform calculations or tasks that are more involved than simple multiplication, division, addition, or subtraction.

12.5.1 Common Math Methods

The Math object contains over 30 methods. The table below provides a sample of the most frequently used options. More complete lists can be found here:

1. [W3 Schools Math Reference](#)
2. [MDN Web Docs](#)

To see detailed examples for a particular method, click on its name.

Table 1: Ten Common Math Methods

Method	Syntax	Description
<i>abs</i>	<code>Math.abs(number)</code>	Returns the positive value of <code>number</code> .
<i>ceil</i>	<code>Math.ceil(number)</code>	Rounds the decimal number UP to the closest integer value.
<i>floor</i>	<code>Math.floor(number)</code>	Rounds the decimal number DOWN to the closest integer value.
<i>max</i>	<code>Math.max(x,y,z,...)</code>	Returns the largest value from a set of numbers.
<i>min</i>	<code>Math.min(x,y,z,...)</code>	Returns the smallest value from a set of numbers.
<i>pow</i>	<code>Math.pow(x,y)</code>	Returns the value of <code>x</code> raised to the power of <code>y</code> (x^y).
<i>random</i>	<code>Math.random()</code>	Returns a random decimal value between 0 and 1, NOT including 1.
<i>round</i>	<code>Math.round(number)</code>	Returns <code>number</code> rounded to the nearest integer value.
<i>sqrt</i>	<code>Math.sqrt(number)</code>	Returns the square root of <code>number</code> .
<i>trunc</i>	<code>Math.trunc(number)</code>	Removes any decimals and returns the integer part of <code>number</code> .

12.5.2 Check Your Understanding

Follow the links in the table above for the `floor`, `random`, `round`, and `trunc` methods. Review the content and then answer the following questions.

Question

Which of the following returns -3 when applied to -3.87?

- a. `Math.floor(-3.87)`
- b. `Math.random(-3.87)`
- c. `Math.round(-3.87)`
- d. `Math.trunc(-3.87)`

Question

What is printed by the following program?

```
1 let          10  
2  
3
```

- a. A random number between 0 and 9.
 - b. A random number between 0 and 10.
 - c. A random number between 1 and 9.
 - d. A random number between 1 and 10.
-

Question

What is printed by the following program?

```
1 let          10  
2  
3
```

- a. A random number between 0 and 9.
 - b. A random number between 0 and 10.
 - c. A random number between 1 and 9.
 - d. A random number between 1 and 10.
-

12.6 Combining Math Methods

The `Math` methods provide useful actions, but each one is fairly specific in what it does (e.g. taking a square root). At first glance, this might seem to limit how often we need to call on `Math`. However, the methods can be manipulated or combined to produce some clever results.

12.6.1 Random Selection From an Array

To select a random item from the array `happiness = ['Hope', 'Joy', 'Peace', 'Love', 'Kindness', 'Puppies', 'Kittens', 'Tortoise']`, we need to randomly generate an index value from 0 to 7. Since `Math.random()` returns a decimal number between 0 and 1, the method on its own will not work.

The `Math.random` appendix page describes how to generate random integers by combining the `random` and `floor` methods. We will use this functionality now.

Let's define a function that takes an array as a parameter. Since we might not know how many items are in the array, we cannot multiply `Math.round()` by a specific value. Fortunately, we have the `length` property...

Example

```
1 function
2     let
3         return
4
5
6 let           'Hope' 'Joy' 'Peace' 'Love' 'Kindness' 'Puppies' 'Kittens' 'Tortoise'
7 ↵
8 for    0      8
9
10
```

repl.it

Console Output

```
Tortoise
Love
Kindness
Hope
Kittens
Kindness
Love
Hope
```

The happiness array has a length of 8, so in line 2 `Math.floor(Math.random() * arr.length)` evaluates as `Math.floor(Math.random() * 8)`, which generates an integer from 0 to 7. Line 3 then returns a random selection from the array.

12.6.2 Rounding to Decimal Places

The `ceil`, `floor`, and `round` methods all take a decimal value and return an integer, but what if we wanted to round 5.56789123 to two decimal places? Let's explore how to make this happen by starting with a simpler example.

`Math.round(1.23)` returns 1, but what if we want to round to one decimal place (1.2)? We cannot alter what `round` does—it *always* returns an integer. However, we CAN change the number used as the argument.

Let's multiply 1.23 by 10 ($1.23 \times 10 = 12.3$) and then apply the method. `Math.round(12.3)` returns 12. Why do this? Well, if we divide 12 by 10 ($12/10 = 1.2$) we get the result of *rounding 1.23 to one decimal place*.

Combining these steps gives us `Math.round(1.23 * 10) / 10`, which returns the value 1.2.

Let's return to 5.56789123 and step through the logic for rounding to two decimal places:

Step	Description
<code>Math.round(5.56789123 * 100) / 100</code>	Evaluate the numbers in () first: $5.56789123 \times 100 = 556.789123$
<code>Math.round(556.789123) / 100</code>	Apply the <code>round</code> method to 556.789123
<code>557 / 100</code>	Perform the division $557/100 = 5.57$

The clever trick for rounding to decimal places is to multiply the original number by some factor of 10, round the result, then divide the integer by the same factor of 10. The number of digits we want after the decimal are shifted in front of the '.' before rounding, then moved back into place by the division.

Table 2: Rounding to Decimal Places

Decimal Places In Answer	Multiply & Divide By	Syntax
1	10	<code>Math.round(number*10)/10</code>
2	100	<code>Math.round(number*100)/100</code>
3	1000	<code>Math.round(number*1000)/1000</code>
etc.	etc.	etc.

12.6.3 Check Your Understanding

Question

Which of the following correctly rounds 12.3456789 to 4 decimal places?

- a. `Math.round(12.3456789)*100/100`
 - b. `Math.round(12.3456789*100)/100`
 - c. `Math.round(12.3456789*10000)/10000`
 - d. `Math.round(12.3456789)*10000/10000`
-

12.7 Exercises: Objects & Math

At our space base, it is a historic day! Five non-human animals are ready to run a space mission without our assistance! For the exercises, you will use the same five animal objects throughout.

Starter Code

1. Based on the two provided object literals and the following data about the remaining three animals, create the three remaining object literals needed for these exercises.

Table 3: Animal Astronauts

Name	Species	Mass (kg)	Age (years)
Brad	Chimpanzee	11	6
Leroy	Beagle	14	5
Almina	Tardigrade	0.0000000001	1

2. For each animal, add a property called `astronautID`. Each `astronautID` should be a randomly selected number between 0 and 10. The number can be 10, but it cannot be 0, and the numbers cannot repeat.
3. Create an array to store all of the animal objects.
4. For upper management at the space base, we need to print out all of the relevant information about the animal astronauts. For each animal, print to the console the following string: ' is a . They are years old and kilograms. Their ID is .' Fill in the blanks with the appropriate values for each animal.
5. Before these animal astronauts can get ready for launch, they need to take a physical fitness test. Add a `move` method to each animal object. The `move` method will select a random number of steps from 0 to 10 for the animal to take. Race the five animals together. An animal is done with the race when they reach 20 steps or greater. Create a new array to store how many turns it takes each animal to complete the race. Print the race

results, '_____ took _____ turns to take 20 steps.' Fill in the blanks with the animal's name and race result.

HINT: There are a lot of different ways to approach this problem. One way that works well is to see how many iterations of the `move` method it will take for one animal to reach 20 steps. Once that has been done for every animal, you can compare the number of iterations to see which animal was the fastest.

12.8 Studio: Objects & Math

In the exercises, you created objects to store data about the candidates for our animal astronaut corps. For this studio, we provide you with a ready-made set of candidates.

You must create code to:

- A. Select the crew.
- B. Perform critical mission calculations.
- C. Determine the fuel required for launch.

12.8.1 Select the Crew

To access the code for exercise 1, open this [repl.it link](#).

1. Each candidate was assigned an ID number, which is stored in the candidate's data file and in the `idNumbers` array.
 - a. Write a function to select a random entry from the `idNumbers` array. Review the *Combining Math Methods* section if you need a reminder on how to do this.
 - b. Use the function to select three ID numbers. Store these selections in a new array, making sure to avoid repeated numbers. No animal can be selected more than once!
 - c. Use one or more loops to check which animals hold the lucky ID numbers. They will be going on the space mission! Store these animals in a `crew` array.
 - d. Use a template literal to print, '_____, _____, and _____ are going to space!' Fill in the blanks with the names of the selected animals.

Tip

`arrayName.includes(item)` can be used to check if the array already contains `item`. A `while` loop can keep the selection process going until the desired number of entries have been added to the array.

12.8.2 Orbit Calculations

To access the code for exercises 2 - 4, go to [repl.it](#).

2. Spacecraft orbits are not circular, but we will assume that our mission is special. The animals will achieve a circular orbit with an altitude of 2000 km.
 - a. Define a function that returns the circumference ($C = 2\pi r$) of the orbit. Round the circumference to an integer.
 - b. Given an orbital speed of 28000 km/hr, calculate how long will it take our animals to complete 5 orbits (time = distance/speed). Round the answer to 2 decimal places.

- c. Print, 'The mission will travel ____ km around the planet, and it will take ____ hours to complete.'
3. Time for an excursion!
 - a. Randomly select one crew member to perform a spacewalk.
 - b. The spacewalk will last for three orbits around the earth. Calculate how many hours the spacewalk will take. Round the answer to 2 decimal places.
 - c. Use the animal's `rate` method to calculate how much oxygen (O_2) they consume during the spacewalk. Round the answer to 3 decimal places.
 - d. Print, '____ will perform the spacewalk, which will last ____ hours and require ____ kg of oxygen.' Fill in the blanks with the animal's name, the spacewalk time, and the amount of O_2 used.

12.8.3 Bonus Missions

Conserve O_2

4. Instead of randomly selecting a crew member for the spacewalk, have your program select the animal with the smallest oxygen consumption rate.

Fuel Required for Launch

To access the code for exercise 5, go to [repl.it](#).

5. A general rule of thumb states that it takes about 9 - 10 kg of rocket fuel to lift 1 kg of mass into low-earth orbit (LEO). For our mission, we will assume a value of 9.5 kg to calculate how much fuel we need to launch our crew into space.
 - a. Write a function that returns the total mass of the selected crew members.
 - b. The mass of the uncrewed rocket plus the food and other supplies is 75,000 kg. Combine the rocket and crew masses, then calculate and store the amount of fuel required to reach LEO.
 - c. Our launch requires a safety margin for the fuel level, especially if the crew members are cute and fuzzy. Add an extra 200 kg of fuel for each dog or cat on board, but only an extra 100 kg for the other species.
 - d. Round the final amount of fuel UP to the nearest integer, then print 'The mission has a launch mass of ____ kg and requires ____ kg of fuel.' Fill in the blanks with the calculated amounts.

MODULES

13.1 What are Modules?

Just like functions should be kept small and accomplish only one thing, we want to apply the same idea for the different parts of our program. **Modules** allow us to keep the features of our program in separate, smaller pieces. We code these smaller chunks and then connect the modules together to create the big project.

Modules are like Legos. Each piece has its own distinct shape and function, and the same set of pieces can be combined in lots of different ways to create unique results.

13.1.1 One Possible Scenario

Imagine we want to create a program that quizzes students on their JavaScript skills.

What would go into this app? Features could include:

1. Selecting questions from a stored array or object.
2. Presenting the questions to the students and collecting their answers.
3. Scoring the quizzes.
4. Storing the results.

This would be a useful app, but we could make it better by adding some other features. Instead of just quizzing students, maybe we could add some tutorial pages. Our app now provides some teaching and assessment content.

Next, how about adding some non-graded practice to make sure the students are ready for their final quiz? Once we accomplish that, we could continue adding to our app to make it better and better.

Let's pause a moment to consider what happened to the size of our project. As the program evolved from the straightforward quiz app to one that included tutorials and practice tasks, the number of lines of code increased. Now imagine we replicate all of these features for two or three other programming languages.

We can picture our app as follows:

The result is a mammoth program that contains thousands of lines of code. How would this impact debugging? How about keeping the code DRY? Do any of the features overlap? How easy is it to add new features?

13.1.2 Why Use Modules

Modules help us keep our project organized. If we find a bug in the quiz part of our program, then we can focus our attention on the quiz module rather than the entire codebase.

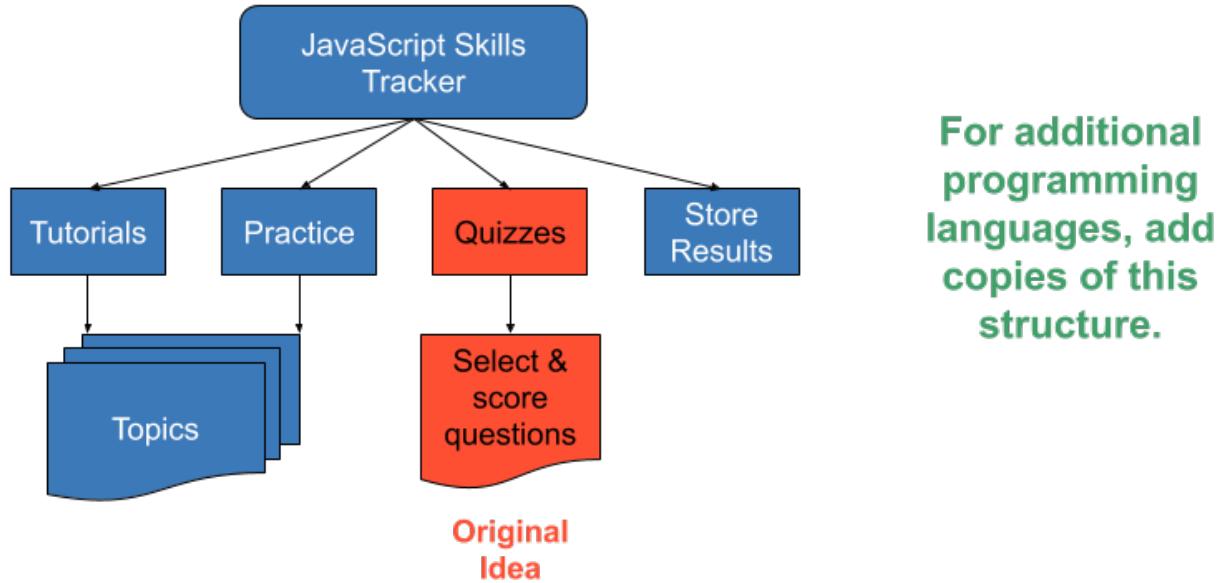


Fig. 1: Imagine if the project included two or three additional programming languages!

Modules also save us effort in other projects - another example of the DRY concept. We have already practiced condensing repetitive tasks into loops or functions. Similarly, if we design our quiz module in a generic way, then we can use that same module in other programs.

Even better, we can **SHARE** our modules with other programmers and use someone else's work (with permission) to enhance our own. Writing the imaginary quiz/tutorial/practice app from scratch would take us many, many weeks. However, someone in the coding community might already have modules that we can immediately incorporate into our own project—saving us time and effort.

Modules keep us from reinventing the wheel.

Some modules also provide us with useful shortcuts. `readline-sync` allowed us to collect input from a user, and this module contains lots of other methods besides the `.question` we used in our examples. Rather than making every developer write their own code for interacting with the user through the console, `readline-sync` makes the process easier for all by providing a set of ready-to-use functions. We do not need to worry about HOW the module works. We just need to be able to pull it into our projects and use its functions.

13.2 Require Modules

In order to take advantage of modules, we must *import* them with the **require** command. You have seen this before with `readline-sync`.

```
1 const readlineSync = require('readline-sync')
2
3 let name = readlineSync.question("What is your name?")
4
5 console.log(`Hello, ${name}`)
```

Line 1 imports the `readline-sync` module and assigns its functions to the `input` variable.

Modules are either *single functions* or *objects that contain multiple functions*. If importing a module returns a single function, we use the variable name to call that function. If the module returns an object, we use dot notation to call the

functions stored in the object. In line 3, we see an example of this. `input.question` calls the `question` function stored in the `readline-sync` module.

Later, we will see examples of importing and using single function modules.

Example

Let's check the type of `input` after we import the `readline-sync` module.

```
const readlineSync = require('readline-sync')
typeof readlineSync
```

Console Output

```
object
```

The `readline-sync` module contains several key/value pairs, each of which matches a key (e.g. `question`) with a specific function.

13.2.1 Where Do We Find Modules?

Modules come from three places:

1. A local file on your computer.
2. Node itself, known as Core modules.
3. An external registry such as NPM.

13.2.2 How Does Node Know Where to Look?

The string value passed into `require` tells Node where to look for a module.

User Created Modules

If a module is stored on your computer, the string passed into `require` must provide a *path* and a *filename*. This path tells Node where to find the module, and it describes how to move up and down within the folders on your computer. Paths can be extremely detailed, but best practice recommends that you keep local modules either in the same folder as your project or only one level from your project. Simple paths are better!

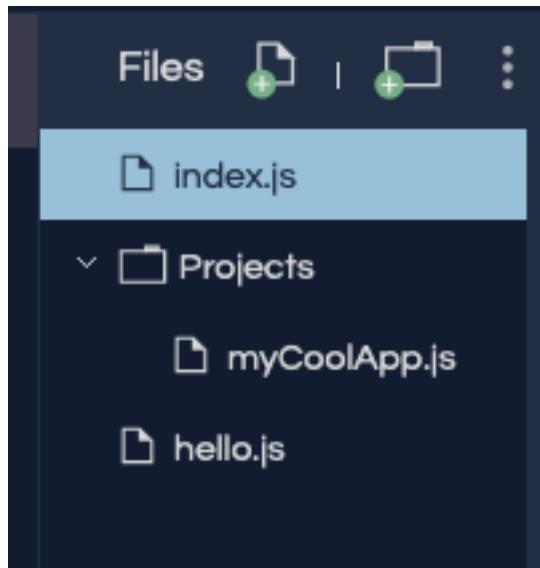
A **relative path** starts with `.` / or `..` /.

1. `.` / tells Node, *Search for the module in the current project folder.*
2. `..` / tells Node, *Search for the module in the folder one level UP from the project.*

As an example, let's assume we have a folder structure like:

Following best practice gives us three scenarios for importing one file into another:

1. **The module is in the same folder:** If we want to import `hello.js` into `index.js`, then we add `const hello = require('./hello.js');` on line 1 of `index.js`.
2. **The module is one level up:** If we want to import `hello.js` into `myCoolApp.js`, then we add `const hello = require('../hello.js');` on line 1 of `myCoolApp.js`.



3. **The module is one level down:** If we want to import `myCoolApp.js` into `index.js`, then we add `const coolApp = require('./Projects/myCoolApp.js');` on line 1 of `index.js`. This tells Node to search for `myCoolApp.js` in the `Projects` sub-folder, which is in the same folder as `index.js`.

Other Modules

If the filename passed to `require` does NOT start with `./` or `../`, then Node checks two resources for the module requested.

1. Node looks for a Core module with a matching name.
2. Node looks for a module installed from an external resource like NPM.

Core modules are installed in Node itself, and as such do not require a path description. These modules are *local*, but Node knows where to find them. Core modules take precedence over ANY other modules with the same name.

Note

[W3 schools](#) provides a convenient list of the Core Node modules.

If Node does find the requested module after checking Core, it looks to the [NPM registry](#), which contains hundreds of thousands of free code packages for developers.

In the next section, we will learn more about NPM and how to use it.

13.2.3 Package.json File

Node keeps track of all the modules you import into your project. This list of modules is stored inside a `package.json` file.

For example, if we only import `readline-sync`, the file looks something like:

```
1  "main"  "index.js"  
2  "dependencies"  
3
```

(continues on next page)

(continued from previous page)

```
4   "readline-sync"  "1.4.9"  
5  
6
```

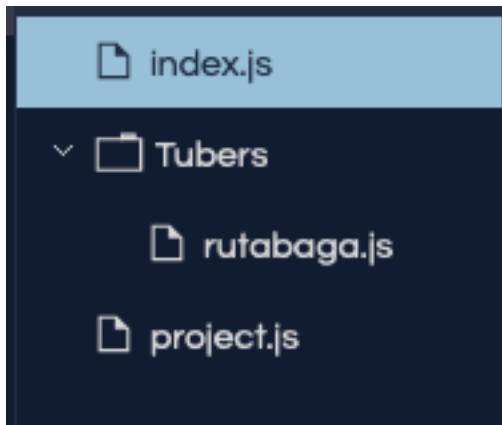
Note

You may not have seen package.json yet, because repl.it hides this file by default. We will talk more about this later.

13.2.4 Check Your Understanding

Question

Assume you have the following file structure:



Which statement allows you to import the rutabaga module into project.js?

- a. const rutabaga = require('/rutabaga.js');
 - b. const rutabaga = require('../rutabaga.js');
 - c. const rutabaga = require('..../rutabaga.js');
 - d. const rutabaga = require('../Tubers/rutabaga.js');
-

13.3 NPM

NPM, Node Package Manager, is a tool for finding and installing Node modules. NPM has two major parts:

1. A registry of modules.
2. Command line tools to install modules.

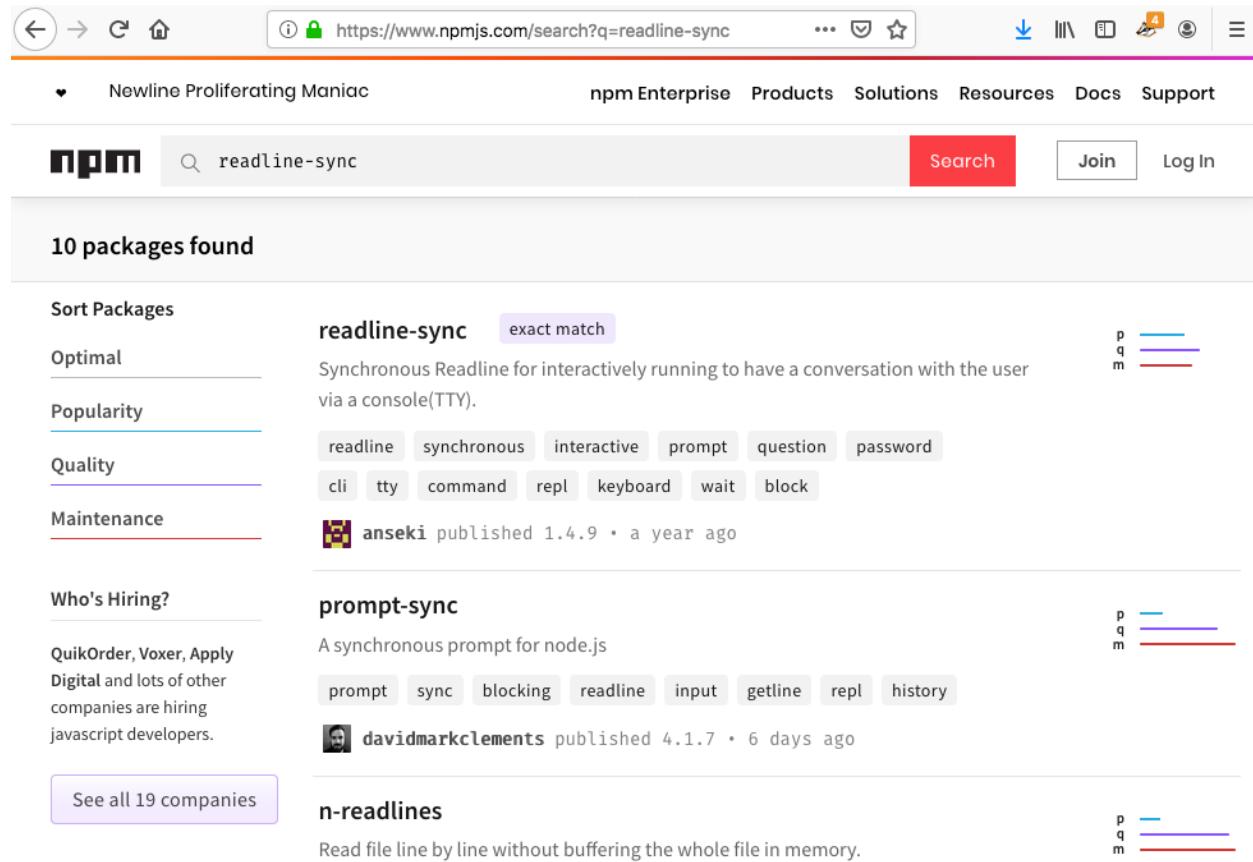
13.3.1 NPM Registry

The NPM registry is a listing of thousands of modules that are stored on a remote server. These can be required and downloaded to your project. The modules have been contributed by other developers just like you.

There is an [online version of the registry](#) where you can search for a module by name or desired functionality.

Example

Go to [online NPM registry](#) and enter “readline-sync” into the search packages input box.



The screenshot shows the npmjs.com search interface. The URL in the address bar is <https://www.npmjs.com/search?q=readline-sync>. The search bar contains "readline-sync". Below the search bar, it says "10 packages found". The first result is "readline-sync" (exact match), which is described as "Synchronous Readline for interactively running to have a conversation with the user via a console(TTY)". It has tags: readline, synchronous, interactive, prompt, question, password, cli, tty, command, repl, keyboard, wait, block. It was published by anseki 1.4.9 a year ago. The second result is "prompt-sync", described as "A synchronous prompt for node.js", published by davidmarkclements 4.1.7 6 days ago. The third result is "n-readlines", described as "Read file line by line without buffering the whole file in memory". On the left, there's a sidebar with sorting options: Optimal (selected), Popularity, Quality, Maintenance, and a "Who's Hiring?" section listing QuikOrder, Voxer, Apply, and Digital. A button "See all 19 companies" is also visible.

An exact match appears as the first result. That is the `readline-sync` module we required. Clicking on the first result leads to the NPM page that describes the `readline-sync` module.

On the details page you will see:

1. Usage statistics (how often the module is used)
 2. Instructions on how to use the module (example code)
 3. Version information
 4. The author(s)
 5. Sourcecode repository
-

The screenshot shows the npmjs.com website interface. At the top, there's a navigation bar with links for 'npm Enterprise', 'Products', 'Solutions', 'Resources', 'Docs', and 'Support'. Below the navigation is a search bar with the placeholder 'Search packages' and a 'Search' button. To the right of the search bar are 'Join' and 'Log In' buttons. The main content area displays the package 'readline-sync'.

readline-sync

1.4.9 • Public • Published a year ago

Readme 0 Dependencies 1,744 Dependents 64 Versions

readlineSync

npm v1.4.9 issues 0 open dependencies No dependency license MIT

Synchronous Readline for interactively running to have a conversation with the user via a console(TTY).

readlineSync tries to let your script have a conversation with the user via a console, even when the input/output stream is redirected like `your-script <foo.dat >bar.log`.

Basic Options Utility Methods Placeholders

- Simple case:

```

var readlineSync = require('readline-sync');

// Wait for user's response.
var userName = readlineSync.question('May I have your name?');
console.log('Hi ' + userName + '!');
    
```

install
`> npm i readline-sync`

weekly downloads
 161,817

version license
 1.4.9 MIT

open issues pull requests
 0 0

homepage repository
github.com

last publish
 a year ago

collaborators

13.3.2 NPM Command Line Interface (CLI)

The NPM command line tool, **CLI**, is installed with Node. The NPM CLI is used in a computer's terminal to install modules into a Node project. Before we can talk more about the NPM CLI, however, we need to discuss repl.it and NPM.

We have coded many Node projects inside of repl.it. Repl.it is great. It allows us to simulate a development environment WITHOUT having to install any software on our computers. As such, it automatically handles much of the work for installing external modules.

Let's examine how using the CLI is different when using repl.it.

13.3.3 CLI With Local Development Environment

The following describes the steps for working with NPM outside of repl.it. Do not worry about following along. We are simply reviewing these to familiarize you with the general process.

1. [Install Node on your computer](#), which also installs the NPM CLI tool.
2. Use the CLI tool in a terminal to install modules into your local project. The syntax is `npm install <package_name>`, and running it downloads the module to your computer and adds an entry into a `package.json` file. This entry indicates that your project depends on the module called `package_name`.
3. More detailed instructions can be found in the [NPM documentation](#).

13.3.4 NPM CLI With repl.it

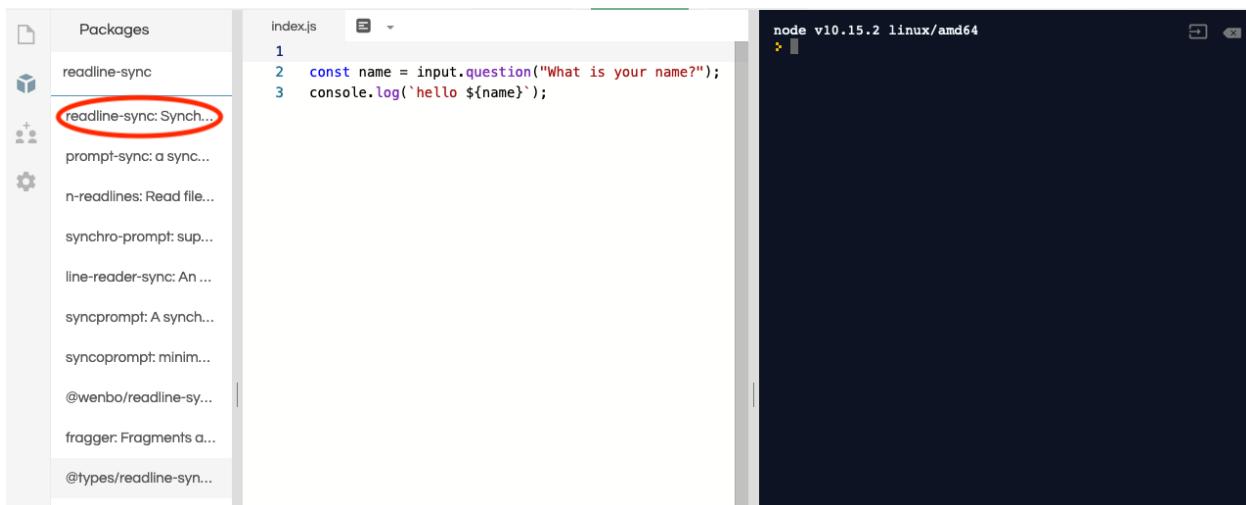
NOW you can follow along, because you use repl.it frequently.

Example

Fork this [example repl.it](#).

Since we are in repl.it, we can skip NPM CLI. Instead, we will use the repl.it interface to add the modules we want.

1. Click on the Packages icon in the left menu (it looks like a box).
2. Enter “readline-sync” in the search box.
3. Click on the top matching result.

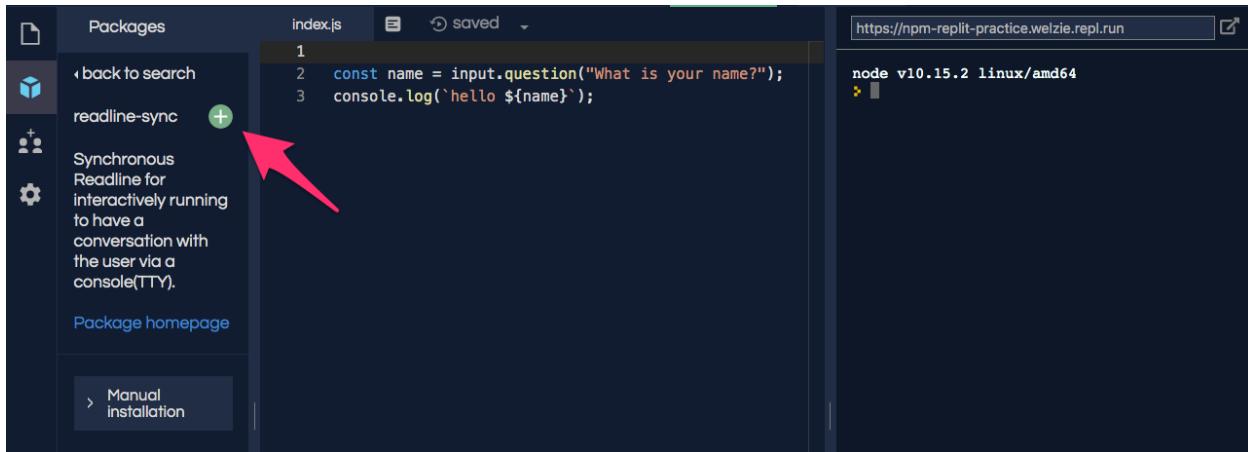


The screenshot shows the repl.it interface. On the left, there is a sidebar titled "Packages" with a list of available modules. One module, "readline-sync", is highlighted with a red circle around its name. The main area contains a code editor with the following code:

```
index.js
1 const name = input.question("What is your name?");
2 console.log(`Hello ${name}`);
```

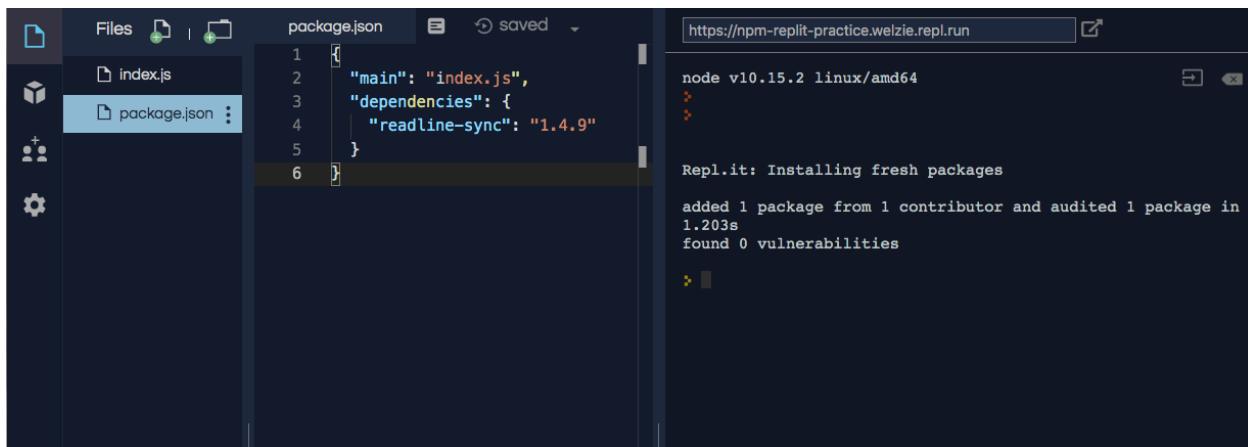
To the right of the code editor is a terminal window showing the command "node v10.15.2 linux/amd64".

1. Verify this is the module you want, then click on the plus icon.



The screenshot shows the npm repl.it interface. On the left, there's a sidebar titled "Packages" with a search bar and a list of packages. The "readline-sync" package is selected, highlighted with a blue background. To its right is a preview area showing the contents of "index.js". Below the preview is a browser window displaying the URL <https://npm-replit-practice.welzie.repl.run>. The browser output shows Node.js version information and a blank command line prompt.

Clicking the plus icon adds a package.json file that includes a dependency listing for readline-sync.



The screenshot shows the npm repl.it interface after adding the readline-sync package. The "Files" sidebar now includes a "package.json" file. The code editor shows the following package.json content:

```

1 {
2   "main": "index.js",
3   "dependencies": {
4     "readline-sync": "1.4.9"
5   }
6 }

```

To the right, a browser window shows the URL <https://npm-replit-practice.welzie.repl.run>. The browser output shows the Node.js version and a log message indicating the package is being installed and found 0 vulnerabilities.

Even though we added readline-sync to package.json, our code still fails because input is not defined. The final step of requiring readline-sync is to assign it to a variable.

Add `const input = require("readline-sync");` to line 1.

```

1 const           "readline-sync"
2
3 const           "What is your name?"
4   `hello ${      }`
```

Note

So far, we used repl.it without a package.json file. That worked because repl.it tries to make the development experience as easy as possible. It hides some details in order to let us pay more attention to our code.

13.4 Exporting Modules

We learned how to pull in useful code in the form of *modules*, but what if we write clever code that we want to share? Fortunately, Node allows us to make our code available for use in other programs.

First, some basic points:

1. Every Node.js file is treated as a module (also called a *package*).
2. From a file, we can export a single function or a set of functions.

13.4.1 Starter Code

We will use the following code sample to practice how to **export** our work—making it available to import as a module.

```
1  function
2      return
3
4
5  function
6      if      2  0
7      return "Even"
8  else
9      return "Odd"
10
11
12
13 function
14     let
15     return
```

repl.it

These functions are in the `practiceExports.js` file, and our goal is to import them into `index.js`.

13.4.2 Exporting a Single Function

Let's start by exporting the `isPalindrome` function. At the bottom of the `practiceExports.js` code, add the line `module.exports = isPalindrome;`. This makes the function available to other files.

In `index.js`, we import `practiceExports.js` with a `require` statement. `isPalindrome` gets pulled in and assigned to the new variable `palindromeCheck`, and we can now call the function from within `index.js`.

Try It

Add the following code to `index.js`, then click “Run”.

```
1  const           './practiceExports.js'
2
3      typeof
4          'that'
5          'radar'
```

Console Output

```
function
false
true
```

There are several points to make about the code and output.

1. By setting `module.exports` equal to `isPalindrome`, we exported that single function.
2. Even though we require the file `practiceExports.js`, it only assigns `isPalindrome` to the variable `palindromeCheck`. Thus, `typeof palindromeCheck` returns `function`.
3. `palindromeCheck` now behaves in the same way as `isPalindrome`, so calling `palindromeCheck('that')` evaluates to `false`, since 'that' is not a palindrome.

13.4.3 Exporting Multiple Functions

`practiceExports.js` contains three functions, and to export all of them we use a different syntax for `module.exports`. Instead of setting up a single function, we will create an *object*.

To export multiple functions, the syntax is:

```
{  
  key1: value1,
  key2: value2,
  key3: value3
}
```

Within the {}, we create a series of key:value pairs. The *keys* will be the names used in `index.js` to call the functions. The *values* are the functions themselves.

Note

We do not have to make the key match the name of the function, but doing so helps maintain consistency between files.

Warning

You might be tempted to use three statements to export the three functions:

```
module.exports = {
  key1: value1,
  key2: value2,
  key3: value3
}
```

This will NOT work, because Node expects only ONE `module.exports` statement in a file. No error will be thrown if you use more than one, but `require('./practiceExports.js')` will only pull in the information from the LAST statement.

Try It

Use the object syntax *as shown above* to modify `module.exports` in `practiceExports.js`. We could include only one or two of the functions, but for this practice let's use all of them.

Next, modify `index.js` as follows and click "Run":

```
1 const           './practiceExports.js'  
2  
3     typeof  
4
```

`typeof` indicates that `practice` is an object, and printing `practice` gives us a list of its key/value pairs (e.g. `isPalindrome: [Function: isPalindrome]`).

All of the functions from `practiceExports` are included in the `practice` object. To call them, we use dot notation—`practice.functionName(argument)`.

Modify `index.js` again and click “Run”:

```
1 const           './practiceExports.js'  
2 let      'Hello'  'World'  123  987  'LC101'  
3  
4             'mom'  
5             9  
6  
7 for    0      3  
8  
9
```

Console Output

```
true  
Odd  
123  
World  
LC101
```

Success! You exported your first module.

13.4.4 What If

You might be wondering, *If I have 20+ functions in a file, and I want to export them ALL, do I really need to type 20+ key/value pairs in `module.exports`?*

The quick answer is, *Yes*. `require` only pulls in items identified in `module.exports`. The longer answer is, *Hmmm, you missed the point*.

Just like functions, we want to keep modules small and specific. Each module should focus on a single idea and contain only a few related functions. With this in mind, we see that `practiceExports` falls short of the goal. Even though it is small in size, `isPalindrome`, `evenOrOdd`, and `randomArrayElement` do not really compliment each other. They would be better placed in different modules.

If you find yourself writing lots of functions in a single file, consider splitting them up into smaller, more detailed modules. Doing this makes debugging easier, organizes your work, and helps you identify which modules to import into a new project. A module titled `cleverLC101Work` is not nearly as helpful as one called `arraySortingMethods`.

13.4.5 Check Your Understanding

Question

A module in Node.js is:

- a. A file containing JavaScript code intended for use in other Node programs.
 - b. A separate block of code within a program.
 - c. One line of code in a program.
 - d. A function.
 - e. A file that contains documentation about functions in JavaScript.
-

Question

Assume you have the following at the end of a `circleStuff.js` module:

Inside your project, you import `circleStuff`:

```
const           './circleStuff.js'
```

Which of the following is the correct way to find the circumference of a circle from within your project?

- a. `circleStuff(argument)`
 - b. `circleStuff.circumference(argument)`
 - c. `circleStuff(circumference(argument))`
 - d. `circumference(argument)`
-

13.5 Wrap-up

In this chapter, we showed how to use `require` to pull a module into your project, and we presented two ways to use `module.exports`. Of course, these are not the only ways to share content.

A quick search online shows that besides functions, we can also share individual variables. There are also alternative syntaxes for `module.exports` - even one that exports as an object, but imports as a function (which means no dot notation).

The skills you practiced in this chapter provide a solid foundation for modules. Learning the alternatives becomes a matter of personal preference and the requirements for your job.

13.6 Exercises: Modules

Practice makes better. You will create a project that accomplishes the following:

- a. Steps through a list of Yes/No questions.
- b. Calls functions based on the user's responses.

Rather than coding all of the functions from scratch, you are going to use existing modules to help assemble your project.

Open [this link](#) and fork the starter code, then complete the following:

13.6.1 Export Finished Modules

Lucky you! Some of your teammates have already coded the necessary functions in the `averages.js` and `display.js` files.

1. In `averages.js`, add code to export all of the functions within an object.
2. In `display.js`, add code to export ONLY `printAll` as a function.

13.6.2 Code & Export New Module

`randomSelect.js` requires your attention.

1. Add code to complete the `randomFromArray` function. It should take an array as an argument and then return a *randomly selected element* from that array.
2. Do not forget to export the `randomFromArray` function so you can use it in your project.

13.6.3 Import Required Modules

The project code is in `index.js`. Start by importing the required modules:

1. Assign `readline-sync` to the `input` variable.
2. Assign the functions from `averages.js` to the `averages` variable.
3. Assign the `printAll` function from `display.js` to the `printAll` variable.
4. Assign the function from `randomSelect.js` to the `randomSelect` variable.

13.6.4 Finish the Project

Now complete the project code. (Note - The line references assume that you added no blank lines during your work in the previous section. If you did, no worries. The comments in `index.js` will still show you where to add code).

1. Line 21 - Call `printAll` to display all of the tests and student scores. Be sure to pass in the correct arguments.
2. Line 24 - Using dot notation, call `averageForTest` to print the class average for each test. Use `j` and `scores` as arguments.
3. Line 29 - Call `averageForStudent` (with the proper arguments) to print each astronaut's average score.
4. Line 33 - Call `randomSelect` to pick the next spacewalker from the `astronauts` array.

13.6.5 Sanity check!

Properly done, your output should look something like:

```
Would you like to display all scores? Y/N: y
Name      Math     Fitness   Coding    Nav       Communication
Fox        95       86         83        81        76
Turtle     79       71         79        87        72
Cat        94       87         87        83        82
Hippo      99       77         91        79        80
Dog        96       95         99        82        70

Would you like to average the scores for each test? Y/N: y
Math test average = 92.6%.
Fitness test average = 83.2%.
Coding test average = 87.8%.
Nav test average = 82.4%.
Communication test average = 76%.

Would you like to average the scores for each astronaut? Y/N: y
Fox's test average = 84.2%.
Turtle's test average = 77.6%.
Cat's test average = 86.6%.
Hippo's test average = 85.2%.
Dog's test average = 88.4%.

Would you like to select the next spacewalker? Y/N: y
Turtle is the next spacewalker.
```

13.7 Studio: Combating Imposter Syndrome

There is a widely recognized condition called **Impostor Syndrome**. First described in the 1970s, it refers to a situation where someone doubts their accomplishments, and they fear that their success is the result of “faking it”. People experiencing imposter syndrome ignore external evidence of their skills, and they attribute their success to luck.

At this point in their learning journey, many new coders feel doubt about their prospects. However, they have PLENTY of company—supreme court justice Sonia Sotomayor, Serena Williams, Tom Hanks, and multiple CEOs have all questioned their success.

The struggle is real, and an open conversation often helps.

13.7.1 You CAN

First, a little perspective. Identify which of the following tasks you have already done or know that you can accomplish:

1. Use code to print “Hello, World” to the screen.
2. Define, initialize, change, and use variables.
3. Convert the string '1234' into a number.
4. Construct a `for` loop to repeat a task 100 times.
5. Construct `if/else if/else` statements to decide which of three tasks to perform.
6. Build, modify, and access an array.
7. Design and call a function.
8. Call one function from within another function.
9. Find and fix bugs in a segment of non-working code.

How many of the 9 items listed above did you indicate? There is no ‘passing’ score for this. Whether you checked all 9 or only 1 or 2, simply saying, *I can do that*, means you have more coding skill than the bulk of the world’s population.

Doubt and uncertainty are normal, especially when exploring a new career. However, with the skills you already know, you can legitimately say, *I am a coder*. Combined with the skills you will learn during the rest of the course, there can be no doubt. You ARE NOT pretending.

13.7.2 Discussion

Take a few moments in the studio to consider, share, and discuss the following:

1. Have you ever felt the effects of Imposter Syndrome? When?
2. Have you ever responded to a compliment by diminishing the work that earned you the praise? If so, why did you answer in that way?
3. How do you feel in a test/quiz/studio when someone finishes much earlier than you?
4. What are you most proud of from your time working with LaunchCode?
5. What are your strengths?
6. What gives you confidence?
7. How can you use your effort and strengths to boost your confidence?

13.7.3 Real World Comments

1. “We all have impostor syndrome. Every creative person at the top of their field will admit to it. And, frankly, getting notoriety for what you do only accentuates your feeling of being a fraud... because... you know how stupid you are, how lazy you are.” – *Adam Savage, one of the hosts from Mythbusters*
2. “I thought it was a fluke. I thought everybody would find out, and they’d take it back. They’d come to my house, knocking on the door, ‘Excuse me, we meant to give that to someone else. That was going to Meryl Streep.’” - *Jodie Foster after winning an Oscar for Best Actress*
3. “Very few people, whether you’ve been in that job before or not, get into the seat and believe today that they are now qualified to be the CEO. They’re not going to tell you that, but it’s true.” - *Starbucks CEO Howard Schultz admitting to being insecure*
4. “Feeling a little uncomfortable with your skills is a sign of learning, and continuous learning is what the tech industry thrives on! It’s important to seek out environments where you are supported, but where you have the chance to be uncomfortable and learn new things.” - *Vanessa Hurst, Co-Founder of Girl Develop It*

13.7.4 Helpful Tips

Imposter syndrome is real and common. However, there are things you can do to help boost your confidence:

1. *Acknowledge the thoughts*, especially when you enter a new point in your life. Recognize that your feelings are normal.
2. *Put it into perspective*. You have been in LC101 for 4 - 5 weeks. It is OK if you do not understand everything on Stack Overflow or recognize all the details about the latest technology.
3. *Review your accomplishments*. Think about your life prior to JavaScript when *string, object* and *function* all meant something much simpler. Your learning has been real!
4. *Share with a trusted friend, teacher or mentor*. Other people with more experience can provide reassurance, and they probably felt similar doubts when they started.

5. *Accept compliments.* Luck did not earn you your tech job. There were LOTS of candidates, and you shined enough to set you apart. If someone compliments your effort or the quality of your work, graciously accept.
6. *Teach.* This is a great way to reinforce your learning, and it helps you recognize how much you know.
7. *Remember the power of ‘Yet’.* You are not the master of all skills, of course, but you do know how to learn. With more practice, you will fill in any gaps in your knowledge.

CHAPTER
FOURTEEN

UNIT TESTING

14.1 Why Test Your Code?

Checking your code is part of the development process. Developers rarely write code without verifying it. You are used to debugging programs as you write them. In fact, we devoted an entire chapter to *debugging* early in the course.

Your development process probably looks something like this:

1. Write code
2. Run program
3. Notice error and investigate
4. Repeat these steps until there are no more errors

But there's a better way to test your code, using *automated* tests. Automated tests actively test your code and help to remove the burden of manual testing. There are many types of automated tests. This chapter focuses on **unit testing**, which tests the smallest components (or *units*) of code. These are typically individual functions.

Before we dive into the *how* of unit testing, let's discuss the *why*.

14.1.1 Know Your Code *Really* Works

Manual testing can eventually lead you to a complete, error-free program. Unit testing provides a better alternative.

This might sound familiar:

You write a program and manually test it. Thinking it is complete, you turn it in only to find that it has a bug or use case that you didn't consider.

The unit testing process helps avoid this by starting with a list of specific, clearly stated behaviors that the program should satisfy. The behaviors are then converted into automated tests that demonstrate program behavior and provide a framework for writing code that *really* works.

14.1.2 Find Regressions

What about this situation?

You write feature #1 for a program. You then move on to feature #2. After finishing feature #2, you realize that your changes broke feature #1.

This is frustrating, right? Especially with larger programs, adding new features often causes unexpected problems in other parts of the code, potentially breaking the entire program. The introduction of such a bug is known as a **regression**.

If you have a collection of tests that can run quickly and consistently, you will know *right away* when a regression appears in your program. This allows you to identify and fix it more quickly.

14.1.3 Tests as Documentation

One of the most powerful aspects of unit testing is that it allows us to clearly define program expectations. A good collection of unit tests can function as a set of *statements* about *how* the program should behave. You and others can read the tests and quickly get an idea of the specifics of program behavior.

Example

Your coworker gives you a function that validates phone numbers, but doesn't provide much detail. Does it handle country codes? Does it require an area code? Does it allow parentheses around area codes? These details would be easily understood if the function had a collection of unit tests that described its behavior.

Code with a good, descriptive set of unit tests is sometimes called **self-documenting code**.

Remembering what your code does and why you structured it a certain way is easy for small programs. However, as the number of your projects increase and their size grows, the need for documentation becomes critical.

Documentation can be in the form of code comments or external text documents. These can be helpful, but have one major drawback which is that they can get out of date very quickly. Out dated, incorrect documentation is very frustrating for a user.

Properly designed unit tests are runnable documentation for your project. Because unit tests are runnable code that declares and verifies features, they can NEVER get out of sync with the updated code. If feature is added or removed, the tests must be updated in order to make them pass.

Let's go ahead and write our first unit test!

14.2 Hello, Jasmine!

In order to unit test our code, we need to use a module. Such a module of called a **unit-testing framework**, **test runner**, or **test harness**, and there are [many to choose from](#).

We will use [Jasmine](#), a popular JavaScript testing framework.

14.2.1 Using Jasmine

Jasmine is an npm module that can be installed like any other npm module via `npm install`. In this chapter we will continue to use repl.it which, automatically installs npm modules when the program is run. For a review of how to install npm modules on your computer see the *Modules chapter*.

Try It!

Run some [tests](#) for the `reverse` function. This is the same `reverse` function that *we wrote previously*.

Don't worry about understanding the code at this point, just hit `run` to execute the tests. How many total tests are there? How many passed? How many failed?

A project using Jasmine has several components. Here's the project structure:

There are three important files:

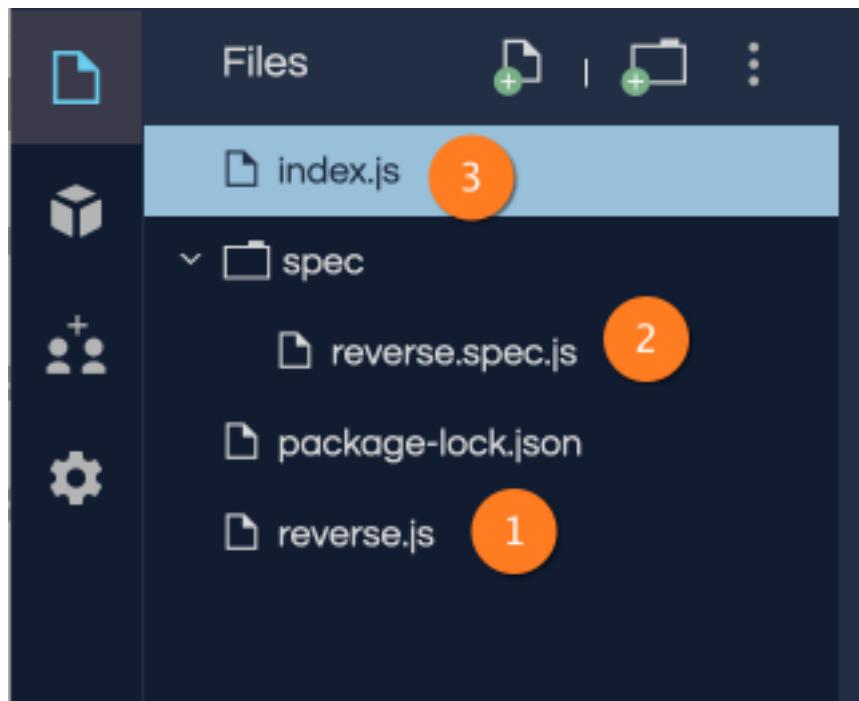


Fig. 1: A Jasmine project

1. `reverse.js` contains the `reverse` function, which must be exported for use in other files.
2. `spec/reverse.spec.js` contains the tests for `reverse`.
3. `index.js` contains the Jasmine code needed to run the tests. This is the file that executes when you hit `run` in repl.it.

Warning

Jasmine can be set up and used in many different ways. If you are looking for an answer on the Internet,—on Stack Overflow or in the Jasmine documentation—you will see widely varying usages of Jasmine that don't apply to your situation. Rely on this book as your main reference, and you'll be fine.

14.2.2 Hello, Jasmine!

Let's build a "Hello, World!" Jasmine project, to get familiar with the basic components. Open and fork [this repl.it project](#)

We will walk you through the steps needed to get a simple Jasmine project up and running. Code along with us throughout this section.

`index.js`

This is the main project file. Up until now, `index.js` is where you have been writing the code for a given exercise or assignment. Now that we are writing tests for our code, `index.js` will contain the Jasmine code to find and execute the tests. Our project-specific code will live in other files.

```
1 const          'jasmine'
2 const        new
3
4
5     'spec'
6
7 "*/[sS]pec.js"
8
9
10
11
```

There are three main components of this program:

1. Lines 1-2 import the Jasmine module and create a new Jasmine object, `jasmine`. This object is responsible for finding and executing our tests.
2. Lines 4-9 configure Jasmine to look for tests in the `spec/` directory of our project. Any file in this directory of the form `fileName.spec.js` will be assumed to contain tests, and will be executed by Jasmine.
3. Line 11 triggers Jasmine to find and execute the tests.

Try It!

Hit `run` on the project. Two things happen:

- repl.it installs Jasmine.
 - Jasmine searches for tests, finding none.
-

Let's add some code to test.

hello.js

If you have not already done so, click `Fork` on the repl.it menu bar so you can edit the starter code.

Create a new file in your project by clicking the icon in the menu bar.



Name the new file `hello.js`, then add this code:

```
1 function
2     if      undefined
3         "World"
4
5     return "Hello, "      "!"
6
```

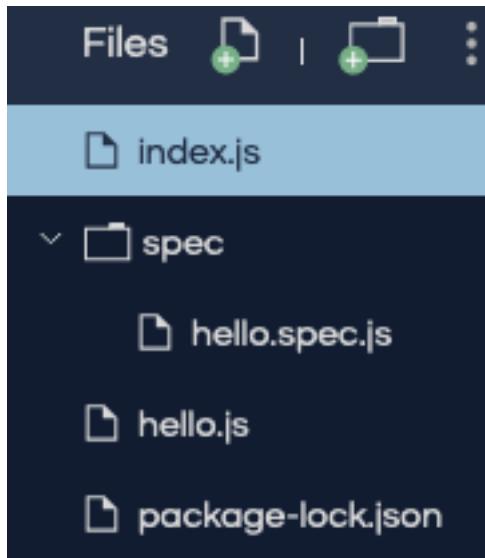
The `hello` function takes a single argument representing a person's name and returns a string greeting that person. If the function is called without an argument, the function returns "Hello, World!".

To use this function outside `hello.js` we must export it. Add this statement at the bottom of the file.

`spec/hello.spec.js`

Now that we have a function to test, let's write some test code. Add a folder named `spec` to the project. Within the folder, create the file `hello.spec.js`. It is conventional to put tests for `fileName.js` in `spec/fileName.spec.js`. This makes it easy to find the tests associated with a given file.

Your file tree should look something like this:



At the top of the `hello.spec.js` file, import your function from `hello.js`, along with the `assert` module:

```
1 const           '../hello.js'  
2 const           'assert'
```

Below that, call the function `describe`, passing in the name of the function we want to test along with an empty anonymous function. `describe` is a Jasmine function that is used to group related tests. Related tests are placed *within* the anonymous function that it receives.

```
"hello"  function
```

Specifications and Assertions

There are two cases we want to test:

1. The function is called with a string argument. In this case, a customized greeting should be returned.
2. The function is called with no argument. In this case, the general greeting should be returned.

Within `describe`'s function argument, place a test for case 1:

```
"should return custom message when name is specified" function
    "Jasmine"    "Hello, Jasmine!"
```

The `it` function is part of the Jasmine framework as well. Calling `it` creates a **specification**, or **spec**, which is a description of expected behavior. The first argument to `it` is a string describing the expected behavior. This string serves to document the test and is also used in reporting test results. Your expectation strings will usually begin with “should”, followed by an expected action.

The second argument to `it` is yet another anonymous function. This function contains the test code itself, which takes the form of an **assertion**. An assertion is a declaration of expected behavior *in code*. Let’s examine the contents of the anonymous function:

```
"Jasmine"    "Hello, Jasmine!"
```

Calling `assert.strictEqual` with two arguments declares that we expect the two arguments to be (strictly) equal. As you get started with unit testing, nearly *all* of your tests will take this form. The first argument to `assert.strictEqual` is a call to the function `hello`. The second argument is the expected output from that function call.

If the two arguments are indeed equal, the test will pass. Otherwise, the test will fail. In this case, we are declaring that `hello("Jasmine")` should return the value `"Hello, Jasmine!"`.

Note

Jasmine also has a `.equal` comparison, which tests for *loose* equality. The difference between loose and strict equality with Jasmine is the same as that of *JavaScript in general*. For this reason, we prefer `.strictEqual` over `.equal`.

Your test file should now look like this:

```
1 const          '../hello.js'
2 const          'assert'
3
4     "hello world test"  function
5
6     "should return a custom message when name is specified" function
7         "Jasmine"    "Hello, Jasmine!"
```

Test Reporting

This is a fully-functioning test file. Hit `run` to see for yourself. If all goes well, the output will look like this:

```
1 Randomized with seed 00798
2 Started
3 .
4
5 1 spec, 0 failures
6 Finished in 0.016 seconds
7 Randomized with seed 00798 (jasmine --random=true --seed=00798)
```

The most important line in the output is this one:

```
1 spec, 0 failures
```

It tells us that Jasmine found 1 test specification, and that 0 of the specs failed. In other words, *our test passed!* The third line also contains useful information. It will contain one dot (.) for each successful test, and an F for each failed test. As our test suite grows, this becomes a nice visual indicator of the status of our tests.

Let's see what a test failure looks like. Go back to `hello.js` and remove the "!" from the return statement:

```
return "Hello, "
```

Run the tests again. This time, the output looks quite different:

```

1 Randomized with seed 98738
2 Started
3 F
4
5 Failures:
6 1) hello world test should return a custom message when name is specified
7 Message:
8     AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
9     + expected - actual
10
11    - 'Hello, Jasmine'
12    + 'Hello, Jasmine!'
13 Stack:
14     error properties: Object({ generatedMessage: true, code: 'ERR_ASSERTION', actual:
15     ↪'Hello, Jasmine', expected: 'Hello, Jasmine!', operator: 'strictEqual' })
16         at <Jasmine>
17         at UserContext.<anonymous> (/home/runner/spec/reverse.spec.js:23:14)
18         at <Jasmine>
19         at runCallback (timers.js:705:18)
20         at tryOnImmediate (timers.js:676:5)
21         at processImmediate (timers.js:658:5)
22
23 1 specs, 1 failure
Finished in 0.021 seconds

```

We intentionally made a test fail. The failing test appears in the `Failures:` section on line 5. This describes exactly what went wrong. The test expected the value '`Hello, Jasmine!`' but received '`Hello, Jasmine`'. Notice that the failure description is the result of joining the two string arguments from `describe` and `it`. This is why we intentionally defined those strings the way we did.

The `Stack:` section on line 13 can be mostly ignored for now. Line 22 has a key statistic showing how many tests, called `specs`, were run and how many failed `1 specs, 1 failure`.

Put `hello.js` back as it was and run the tests again to make sure it works.

Let's add a final spec to test our other case.

```
"should return a general greeting when name is not specified" function
    "Hello, World!"
```

This spec declares that calling `hello()` should return "`Hello, World!`". Run the tests again and you'll see this output:

```
Randomized with seed 81081
Started
..
2 specs, 0 failures
Finished in 0.025 seconds
Randomized with seed 81081 (jasmine --random=true --seed=81081)
```

Nice work! You just created your first program with a full test suite. You can view [our full Hello, Jasmine! project](#) for reference.

There are a lot of details in the setup of these tests, so take a few minutes to look over the code and describe to yourself what each component is doing.

Note

There are many ways to structure test specifications. If you look at the official Jasmine documentation, you'll see specs with different code in place of `assert.strictEqual`:

We have chosen to use `assert.strictEqual` because its syntax is more similar to common testing frameworks in other languages like Java and C#. Learning to use `assert.strictEqual` will make it easier for you to transition to one of those frameworks later in the class.

14.2.3 Check Your Understanding

Question

Examine the function below, which checks if two strings match:

```
1  function
2      if
3          return 'Strings match!'
4      else
5          return 'No match!'
6
7
```

Which of the following tests checks if the function properly handles case-*sensitive* answers.

- `assert.strictEqual(doStringsMatch('Flower', 'Flower'), 'Strings match!');`
 - `assert.strictEqual(doStringsMatch('Flower', 'flower'), 'No match!');`
 - `assert.strictEqual(doStringsMatch('Flower', 'plant'), 'No match!');`
 - `assert.strictEqual(doStringsMatch('Flower', ''), 'No match!');`
-

14.3 Unit Testing in Action

Testing is a bit of an art; there are no hard and fast rules about how to go about writing good tests. That said, there are some general principles that you should follow. In this section, we explore some of these.

In particular, we focus on identifying good **test cases** by working through a specific example. A test case is a single situation that is being tested.

14.3.1 What to Test

When writing tests for your code, what should you test? You can't test *every* possible situation or input. But you also don't want to leave out important cases. A function or program that isn't well-tested might have bugs lurking beneath the surface.

Note

Since we are focused on *unit* testing, in this chapter we will generally use the term “unit” to refer to the function or program under consideration.

Regardless of the situation, there are three types of test cases that you should consider: positive, negative, and edge cases.

1. A **positive test** verifies expected behavior with valid data.
 2. A **negative test** verifies expected behavior with *invalid* data.
 3. An **edge case** is a subset of positive tests, which checks the extreme edges of valid values.
-

Example

Imagine a function named `setTemperature` that accepts a number between 50 and 100.

1. Positive test values: 56, 75, 80
 2. Negative test values: -1, 101, "70"
 3. Edge case values: 50, 100
-

Considering positive, negative, and edge tests will go a long way toward helping you create well-tested code.

Let's see these in action, by writing tests for *our `isPalindrome` function*.

14.3.2 Setting Up

Here's the function we want to test:

```
1  function
2    return ...
3
4
5  function
6    return ...
7
```

repl.it

Code along with us by forking [our repl.it starter code project](#), which includes the above code in `palindrome.js` and the Jasmine test runner code in `index.js`. Note that we have removed the `console.log` statements from the original code and exported the `isPalindrome` function:

Tip

When creating a unit-tested project, *always* start by copying the Jasmine test runner code into `index.js` and putting the code you want to test in an appropriately named `.js` file.

You have become used to testing your code by running it and printing output with `console.log`. When writing unit-tested code, we no longer need to take this approach.

Tip

If you find yourself tempted to add a `console.log` statement to your code, write a unit test instead! You would mostly likely remove that `console.log` after getting your code to work, while the test will remain for you and other developers to use in the future.

Finally, create `spec/` folder and add a spec file, `palindrome.spec.js`. This file should include imports and a `describe` block:

```
1 const           '../palindrome.js'  
2 const           'assert'  
3  
4   "isPalindrome"  function  
5  
6 // TODO - write some tests!  
7  
8
```

Okay, let's write some tests!

14.3.3 Positive and Negative Test Cases

Positive Test Cases

We'll start with positive and negative tests. For `isPalindrome`, some positive tests have inputs:

- `"a"`
- `"aaaa"`
- `"aba"`
- `"racecar"`

Calling `isPalindrome` with these inputs should return `true` in each case. Notice that these tests are as simple as possible. Keeping test inputs simple, while still covering your desired test cases, will make it easier to fix a bug in the event that a unit test fails.

Let's add tests for these inputs to `spec/palindrome.spec.js`:

```
1 const           '../palindrome.js'  
2 const           'assert'  
3  
4   "isPalindrome"  function  
5
```

(continues on next page)

(continued from previous page)

```

6   "should return true for a single letter"  function
7     "a"      true
8
9
10  "should return true for a single letter repeated"  function
11    "aaa"    true
12
13
14  "should return true for a simple palindrome"  function
15    "aba"    true
16
17
18  "should return true for a longer palindrome"  function
19    "racecar"  true
20
21
22

```

Note the clear test case descriptions (for example, “should return true for a single letter repeated”), which will help us easily identify the expected behavior of our code later.

After adding the positive tests to your file, run them to make sure they all pass.

Negative Test Cases

For `isPalindrome`, some negative tests have inputs:

- "ab"
- "launchcode"
- "abA"
- "so many dynamos"

Calling `isPalindrome` with these inputs should return `false` in each case. The last two of these negative tests deserve a bit more discussion.

When writing our `isPalindrome` function initially, we made two important decisions:

- Case should be considered, and
- whitespace should be considered.

The definition of a palindrome differs sometimes on these two matters, so it's important to test them.

Testing with input "abA" ensures that case is considered, since the lowercase version of this string, "aba", is a palindrome. Testing with "so many dynamos" ensures that whitespace is considered, since the version of this string with whitespace removed, "somanydynamos", is a palindrome.

Note

It's important to isolate your test cases. For example, "So Many Dynamos" is a poor choice of input for a negative test, since it contains *two* characteristics that are being tested for - case and whitespace. If a test with this input failed, it would NOT be clear why it failed.

Including specific tests that demonstrate how *our* `isPalindrome` function behaves in these situations helps make our code *self-documenting*. Someone can read our tests and easily see that we *do* consider case and whitespace.

Let's add some test for these negative cases. Add these within the `describe` call.

```
"should return false for a longer non-palindrome" function
    "launchcode"  false

"should return false for a simple non-palindrome" function
    "ab"          false

"should be case-sensitive"   function
    "abA"         false

"should consider whitespace" function
    "so many dynamos"  false
```

Now run the tests to make sure they pass. Your code now includes a set of tests that considers a wide variety of positive and negative cases.

14.3.4 Edge Cases

Recall our definition of **edge case**:

An edge case is a test case that provides input at the extreme edge of what the unit should be able to handle.

Edge cases can look very different for different units of code. Most of the examples we provided above dealt with numerical edge cases. However, edge cases can also be non-numeric.

In the case of `isPalindrome`, the most obvious edge case would be that of the empty string, `" "`. This is the smallest possible string that we can use when calling `isPalindrome`. Not only is it the smallest, but it is essentially *different* from the next longest string, `"a"`—one has characters and one doesn't.

Should the empty string be considered a palindrome? That decision is up to us, the programmer, and there is no right or wrong answer. In our case, we decided to take a very literal definition of the term “palindrome” by considering case and whitespace. In other words, our definition says that a string is a palindrome exactly when it equals its reverse. Since the reverse of `" "` is also `" "`, it makes sense to consider the empty string a palindrome.

Let's add this test case to our spec:

```
"should consider the empty string a palindrome" function
    ""      true
```

Now run the tests, which should all pass.

You might think that another edge case is that of the longest possible palindrome. Such a palindrome would be as long as the longest possible string in JavaScript. This case is not worth considering for a couple of reasons:

- The length of the longest string [can vary across different JavaScript implementations](#).
- The most recent JavaScript specification, ES2016, states that the maximum allowed length of a string should be $2^{53} - 1$ characters. This is a LOT of characters, and it is unrealistic to expect that our function will ever be given such a string.

14.3.5 Toward a Better Testing Workflow

In this case, we had a well-written function to write tests for, so it was straightforward to create tests that pass. Most situations will not be this simple. Your tests will often uncover bugs, forcing you to go back and update your code. That's okay! This is precisely what tests are for.

The workflow for this situation is:

1. Write code
2. Write tests
3. Fix any bugs found while testing

The rest of the chapter focuses on a programming technique that allows you to completely *eliminate* the third step, by reversing the order of the first two:

1. Write tests
2. Write code

As you will soon learn, writing your tests *before* the code is a great way to enhance your programming efficiency and quality.

14.3.6 Check Your Understanding

Let's assume we updated `isPalindrome` to be case-insensitive (e.g. `isPalindrome('Radar')` returns `true`).

Question

Which of the following is an example of *positive* test case for checking if `isPalindrome` is case-insensitive?

- a. aa
 - b. aBa
 - c. Mom
 - d. Taco Cat
 - e. AbAb
-

Question

Which of the *negative* test cases listed above are no longer valid for our case-insensitive `isPalindrome`?

- a. ab
 - b. launchcode
 - c. abA
 - d. so many dynamos
-

14.4 Test-Driven Development

Now that we know more about unit testing, we are going to learn a new way of using them. So far we have written tests to verify functionality of *existing* code. Next we are going to use tests to verify functionality of code that does NOT already exist. This may sound odd, but this process has many benefits as we will learn.

As the name sounds **Test-driven development (TDD)** is a software development process where the unit tests are written first. However that doesn't tell the entire story. Writing the tests first and intentionally thinking more about the code design leads to better code. The name comes from the idea of the tests *driving* the development process.

Before we can start using TDD, we need a list of discrete features that can be turned into unit tests. This will help keep our tests focused on specific functionality which should lead to code that is easy to read. Along the way we will build confidence as we add features.

Note

TDD is a process that some organizations choose to use. Using the TDD process is not required when using unit tests.

14.4.1 The Test/Code Cycle

With TDD you start with the unit test first. As with any unit test, the test should describe a clearly described behavior that can be tested.

Example

Example test case for a data parsing project.

- Take in string of numbers delimited by a character and return an array.
-

Because the test is for a feature that does NOT exist *yet*, we need to think about how the feature will be implemented. This is the time to ask questions like. Should we add a new parameter? What about an entirely new function? What will the function return?

Example

How could we implement our test case? Remember we aren't writing the code yet, only thinking about the design.

- A function named `parseData`
 - `parseData` will
 - have a `data` parameter will a string of data
 - have a `delimiter` parameter will be used to split the array
 - return an array
 - `parseData` will be defined in a module
-

Next, write the unit test as if the parameter or function you imagined already exists. This may seem a bit odd, but considering how the new code will be used helps find bugs and flaws earlier. We also have to use test utilities such `assert.strictEqual` to will clearly demonstrate that the proposed new code functions properly.

Example

This is where the ideas are typed out into a test. In this example the test references a module and a function that have not been created yet. The code follows the plan we came up with earlier. Very importantly there is an `assert.strictEqual` that verifies an array is returned.

```

1 const assert = require('assert');
2 const parseNumbers = require('../parse-numbers');
3
4 "parse numbers" function parseNumbers(str) {
5   "returns array when passed comma separated list of numbers" function() {
6     let result = parseNumbers("5,8,0,17,6,4,9,3");
7     assert.strictEqual(result, [5, 8, 0, 17, 6, 4, 9, 3]);
8   }
9 }
10
11

```

Now run the test! The test should fail or possibly your code will not compile because you have referenced code that does not exist yet.

Finally, write code to pass the new test. In the earlier chapters, this is where you started, but with TDD writing new code is the *last* step.

Example

To make the above test pass a file would be created that exports a `parseData` function with logic that satisfies the expected result.

```

1 function parseData(str) {
2   return str.split(',');
3 }
4
5

```

Coding this way builds confidence in your work. No matter how large your code base may get, you know that each part has a test to validate its functionality.

Example

Now that we have one passing test for our data parser project, we could confidently move on to writing tests and code for the remaining features.

14.4.2 Red, Green, Refactor

While adding new features and making our code work is the main goal, we also want to write readable, efficient code that makes us proud. The red, green, refactor mantra describes the process of writing tests, seeing them pass, and then making the code better. As the name suggests, the cycle consists of three steps. Red refers to test results that fail, while green represents tests that pass. The colors refer to test results which are often styled with red for failing tests and green for passing tests.

1. Red -> Write a failing test.
2. Green -> Make it pass by implementing the code.
3. Refactor -> Make the code better.

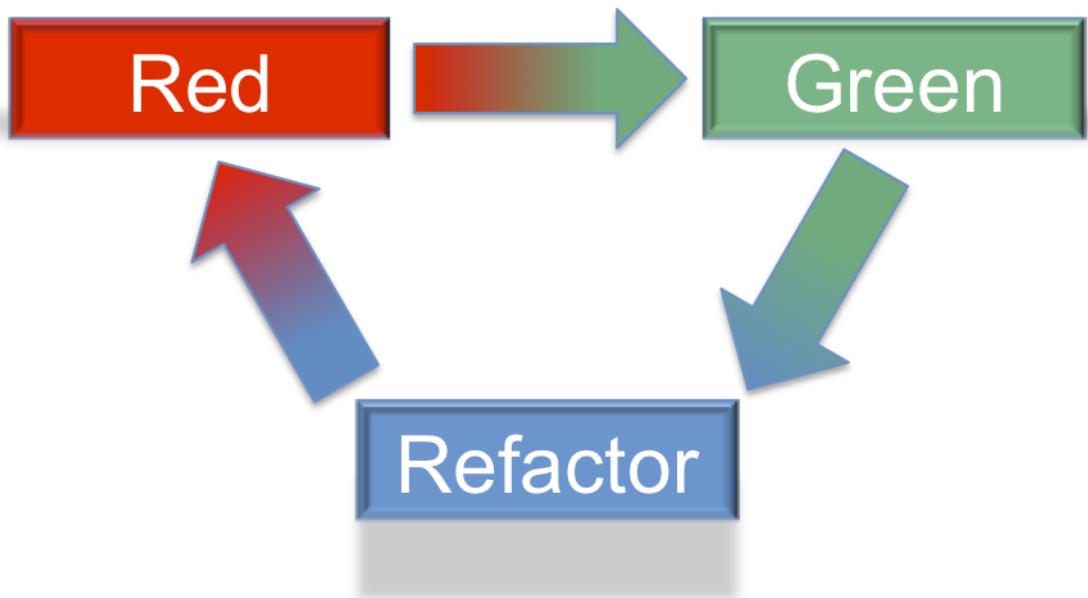


Fig. 2: Red, green, refactor cycle.

Refactoring code means to keep the same overall feature, but change how that feature is implemented. Since we have a test to verify our code, we can change the code with confidence, knowing that any regression will be immediately identified by the test. Here are a few examples of refactoring using different data structures, reducing the number of times needed to loop through an array, or even moving duplicate logic into a function so it can be reused.

The refactor is also done in a TDD process. Decide on what is the improved way of implementing the feature and then change the unit test to use this new idea. See the test fail, then implement the refactor idea. Finally see the tests pass with the refactored design.

14.5 TDD in Action

Fork our starter code [repl.it](#) and follow along as we implement a project using TDD.

We need to write a Node module to process transmissions from the [Voyager1](#) probe.

Example

Transmission

```
"1410::<932829840830053761>"
```

Expected Result

```
1410  
932829840830053761
```

14.5.1 Requirements

The features for this project have already been broken down into small testable units. Let's review them and then we will take it slow, one step at a time.

1. Take in a transmission string and return an object.
2. Return `-1` if the transmission does NOT contain `"::"`.
3. Returned object should contain an `id` property
 - The value of `id` is the part of the transmission *before* the `"::"`
4. The `id` property should be of type `Number`
5. Returned object should contain a `rawData` property
 - The value of `rawData` is the part of the transmission *after* the `"::"`
6. Return `-1` for the value `rawData` if the `rawData` part of the transmission does NOT start with `<` and end with `>`

14.5.2 Requirement #1

Requirement: Take in a transmission string and return an object.

To get started on this we need to:

- a. Create a blank test function.
- b. Give the test a name that is a clear, testable statement.

Creating a blank test is easy, go to `processor.spec.js` and add an empty test method. Tests in Jasmine are declared with an `it` function. Remember that tests go inside of the `describe` function, which along with the string parameter describe the group of tests inside.

```
1 const           'assert'  
2  
3   "transmission processor"  function  
4  
5   ""  function  
6  
7  
8  
9
```

Give the test the name "takes a string returns an object".

```
1 const           'assert'  
2  
3   "transmission processor"  function  
4  
5   "takes a string returns an object"  function  
6  
7  
8  
9
```

Now that we identified a clear goal for the test, let's add logic and `assert` calls in the test to verify the desired behavior. *But wait...* we haven't added anything except an empty at this point. There isn't any actual code to verify. That's okay, this is part of the TDD process.

We are going to think about and visualize how this feature should be implemented in code. Then we will write out in the test how this new code will be used.

We need to think of something that will satisfy the statement `it ("takes a string returns an object")`. The `it` will be a function that is imported from a module. Below on line 2, a `processor` function is imported from the `processor.js` module.

```
1 const assert = require('assert')
2 const processor = require('../processor.js')
3
4 "transmission processor" function
5
6 "takes a string returns an object" function
7
8
9
10
```

We have an idea for a function named `processor` and we have imported it. Keep in mind this function only exists as a concept and we are writing a test to see if this concept makes sense.

Now for the real heart of the test. We are going to use `assert.strictEqual` to verify that if we pass a string to `processor` that an object is returned. Carefully review lines 7 and 8 shown below.

```
1 const assert = require('assert')
2 const processor = require('../processor.js')
3
4 "transmission processor" function
5
6 "takes a string returns an object" function
7 let result = processor("9701::<489584872710>")
8 assert.strictEqual(typeof result, "object")
9
10
11
```

On line 7 the `processor` function is called, with the value being stored in a `result` variable. On line 8 the result of the expression `typeof result` is compared to the value `"object"`. Reminder that the `typeof operator` returns a string representation of a type. If `typeof result` evaluates to the string `"object"`, then we know that `processor` returned an object.

Code Red

Let's run the test! Click the `run >` button in your repl.it. You should see an error about `processor.js` not existing. This makes sense, because we have not created the file yet. We are officially in the Red phase of Red, Green, Refactor!

```
Error: Cannot find module '../processor.js'
```

Go Green!

Now that we have a failing test, we have only one choice. Make it pass.

- a. Add a `processor.js` file to your repl.it.
- b. Inside of the module declare a `processor` function that takes a parameter and returns an object.

Contents of the new `processor.js` file.

```

1 function
2     return
3
4
5

```

```

1 function process(transmission) {
2     return {};
3 }
4
5 module.exports = process;
6

```

Fig. 3: processor.js file

Run the test again.

We did it! 1 spec, 0 failures means 1 passing test. In repl.it you have to imagine the satisfying green color of a passing test.

```

1 spec, 0 failures
Finished in 0.011 seconds

```

Refactor if Needed

This solution is very simple and does not need to be improved. The refactor step does not always lead to an actual changing of your code. The most important part is to review your code to make sure that it's efficient and meets your team's standards.

14.5.3 Requirement #2

Requirement: Return -1 if the transmission does NOT contain " : :".

Next we have a negative test requirement that tells us what should happen if the data is invalid. Before jumping into the code, let's review the steps we took to implement requirement #1.

Review of TDD process:

1. Create a blank test function.
2. Give the test a name that is a clear, testable statement.
3. Come up with test data that will trigger the described behavior.
4. Think about what is needed, then write code that fulfills the stated behavior.
5. Run the test and see it fail.

6. Implement the new code or feature used in the test.
7. Run the test and see it pass.
8. Review to see if refactor needed.

For requirement #2, the solution for *steps 1 - 4* can be seen on lines 11 - 14 below.

```
1 const assert = require('assert')
2 const processor = require('../processor.js')
3
4 "transmission processor" function
5
6 "takes a string returns an object" function
7 let transmissionProcessor = () =>
8     assert.typeof(transmissionProcessor(), 'object')
9
10
11 "returns -1 if '::' not found" function
12 let transmissionProcessor = () =>
13     assert.strictEqual(transmissionProcessor('::'), -1)
14
15
16
```

Now for *step 5*, run the test and see it fail. When you run the tests, you should see the below error message. Notice that `-1` was the expected value, but the actual value was `'object'`.

```
Failures:
1) transmission processor returns -1 if '::' not found
Message:
AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
+ expected - actual
- 'object'
+ -1
```

Next is *step 6*, write code that will make the test pass. Go to `processor.js` and update the `processor` function to check the `transmission` argument for the presence of `:::`.

```
1 function processor(transmission) {
2     if (transmission === '') {
3         // Data is invalid
4         return -1
5     }
6
7     return 0
8 }
9
```

Lucky *step 7* is to run the tests again. They should both pass.

```
2 specs, 0 failures
Finished in 0.035 seconds
```

Finally *step 8* is to review the code to see if it needs to be refactored. As with the first requirement our code is quite simple and can not be improved at this time.

14.5.4 Requirement #3

Requirement: Returned object should contain an `id` property. The `id` is the part of the transmission *before* the " :: ". The same steps will be followed, even though they are not explicitly listed.

See lines 16 - 19 to see the test added for this requirement. To test this case `notStrictEqual` was used, which is checking if the two values are NOT equal. `notStrictEqual` is used to make sure that `result.id` is NOT equal to `undefined`. Remember that if you reference a property on an object that does NOT exist, `undefined` is returned.

```

1 const assert = require('assert')
2 const processor = require('../processor.js')
3
4 "transmission processor" function
5
6 "takes a string returns an object" function
7 let result = "9701::<489584872710>"
8 typeof result === "object"
9
10
11 "returns -1 if '::::' not found" function
12 let result = "9701<489584872710>" + 1
13
14
15 "returns id in object" function
16 let result = "9701::<489584872710>" + undefined
17
18
19
20
21

```

The fail message looks a little different than what we have seen. The phrase “Identical input passed to `notStrictEqual`” lets us know that the two values were equal when we didn’t expect them to be.

```

Failures:
1) transmission processor returns id in object
Message:
AssertionError [ERR_ASSERTION]: Identical input passed to notStrictEqual: undefined

```

The object returned from `processor` doesn’t have an `id` property. We need to split the transmission on ' :: ' and then add that value to the object with the key `id`. See solution in `processor.js` below.

```

1 function processor(str) {
2   if (str === "") return 0
3   // Data is invalid
4   return 1
5
6   let result = str
7   return result
8   0
9
10
11
12

```

Run the tests again. That did it. The tests pass! :-)

Line 6 splits transmission into the parts array, and line 8 assigns the first entry in the array to the key id.

```
3 specs, 0 failures
Finished in 0.011 seconds
```

14.5.5 Requirement #4

Requirement: The id property should be of type Number

Again the same steps are followed, though not listed.

New test to be added to specs/processor.spec.js

```
1   "converts id to a number"  function
2     let                      "9701::<489584872710>" 
3           9701
4
```

Fail Message

```
Failures:
1) transmission processor converts id to a number
Message:
AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
+ expected - actual

- '9701'
+ 9701
```

Convert the id part of the string to be of type number.

```
1  function
2    if                      ":"      0
3      // Data is invalid
4      return 1
5
6    let                      ":"      "
7    return
8          0
9
10
11
12
```

Now for the great feeling of a passing tests!

```
4 specs, 0 failures
Finished in 0.061 seconds
```

Note

You may be wondering what happens if that data is bad and the id can't be turned into a number. That is a negative test case related to this feature and is left for you to address in the final section.

14.5.6 Requirement #5

Requirement: Returned object should contain a rawData property. The rawData is the part of the transmission after the " :: "

New test to be added to specs/processor.spec.js

```

1  "returns rawData in object"  function
2    let                      "9701::<487297403495720912>" 
3                                undefined
4

```

Fail Message

```

Failures:
1) transmission processor returns rawData in object
Message:
AssertionError [ERR_ASSERTION]: Identical input passed to notStrictEqual: undefined

```

We need to extract the rawData from the second half of the transmission string after it's been split. Then return that in the object.

```

1  function
2    if          " ::"      0
3      // Data is invalid
4      return 1
5
6    let          " ::"
7    let          1
8    return      0
9
10
11
12
13
14

```

It's that time again, our tests pass!

```

5 specs, 0 failures
Finished in 0.041 seconds

```

14.5.7 Requirement #6

Requirement: Return -1 for the value rawData if the rawData part of the transmission does NOT start with < and end with >

Let's think about what test data to use for this requirement. What ways could the transmission data be invalid?

1. It could be missing < at the beginning
2. It could be missing > at the end
3. It could be missing both < and >
4. Has < but is in the wrong place
5. Has > but is in the wrong place

Introduction to Professional Web Development in JavaScript

All these cases need to be covered by a test. Let's start with #1, which is missing < at the beginning.

New test to be added to specs/processor.spec.js

```
1 "returns -1 for rawData if missing < at position 0" function
2 let "9701::487297403495720912>" 1
3
4
```

Fail Message

```
Failures:
1) transmission processor returns -1 for rawData if missing < at position 0
Message:
AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
+ expected - actual
- '487297403495720912'
+ -1
```

New code added to processor.js to make tests pass. Note that we don't simply return -1, the requirement is to return the object and set the value of rawData to -1.

```
1 function
2   if ":" 0
3     // Data is invalid
4     return 1
5
6   let ":" 1
7   let 0 "<"
8   if 0 "<" 1
9     1
10
11  return 0
12
13
14
15
16
17
```

You know what's next, our tests pass!

```
6 specs, 0 failures
Finished in 0.056 seconds
```

Try It!

The test data we used was missing < at the beginning. Add tests to cover these cases. -1 should be returned as the value for rawData for all of these.

- "9701::8729740349572>0912"
 - 9701::4872<97403495720912"
 - 9701::487297403495720912"
 - 9701::<487297403495<720912>"
-

14.5.8 Use TDD to Add These Features

Use the steps demonstrated above to implement all or some of the below features. Take your time, you can do it!

1. Trim leading and trailing whitespace from `transmission`.
2. Return `-1` if the `id` part of the `transmission` can not be converted to a number.
3. Return `-1` if more than one `"::"` found in `transmission`
4. Return `-1` for value of `rawData` if anything besides numbers are present
5. Allow for multiple `rawData` values
 - `rawData` would be returned as an array of numbers
 - Get the new test working and then fix any broken existing tests
 - Example Transmission: `"9701::<21212.232323.242424>"`
 - Result: `{ id: 9701, rawData: [21212, 232323, 242424] }`

14.6 Exercises: Unit Testing

In many of your previous coding tasks, you had to verify that your code worked before moving to the next step. This often required you to add `console.log` statements to your code to check the value stored in a variable or returned from a function. This approach finds and fixes syntax, reference, or logic errors AFTER you write your code.

In this chapter, you learned how to use unit testing to solve coding errors. Even better, you learned how to PREVENT mistakes by writing test cases before completing the code. The exercises below offer practice with using tests to find bugs, and the studio asks you to implement TDD.

14.6.1 Automatic Testing to Find Errors

Let's begin with the following, simple code:

```
1  function
2    let      ''
3    if      5
4      " is less than 5."
5    else if   5
6      " is equal to 5."
7    else
8      "is greater than 5."
9
10
11   return
```

repl.it

The function checks to see if a number is greater than, less than, or equal to 5. We do not really need a function to do this, but it provides good practice for writing test cases.

Note that the repl.it contains three files:

- a. `checkFive.js`, which holds the code for the function,
- b. `checkFive.spec.js`, which will hold the testing code,

- c. `index.js` which holds special code to make Jasmine work.

Warning

Do NOT change the code in `index.js`. Messing with this file will disrupt the automatic testing.

1. We need to add a few lines to `checkFive.js` and `checkFive.spec.js` to get them to talk to each other.
 - a. `checkFive.spec.js` needs to access `checkFive.js`, and we also need to import the `assert` testing function. Add two `require` statements to accomplish this (review *Unit Testing in Action* if needed).
 - b. Make the `checkFive` function available to the spec file, by using `module.exports` (review *Unit Testing in Action* if needed).
2. Set up your first test for the `checkFive` function. In the `checkFive.spec.js` file, add a `describe` function with one `it` clause:

```
const          '../checkFive.js'  
const          'assert'  
  
"checkFive"  function  
  
"Descriptive feedback..."  function  
//code here...
```

3. Now write a test to see if `checkFive` produces the correct output when passed a number *less than 5*.
 - a. First, replace `Descriptive feedback...` with a DETAILED message. This is the text that the user will see if the test *fails*. Do NOT skimp on this. Refer back to the *Specifications and Assertions* section to review best practices.
 - b. Define the variable `output`, and initialize it by passing a value of 2 to `checkFive`.

```
const          '../checkFive.js'  
const          'assert'  
  
"checkFive"  function  
  
"Descriptive feedback..."  function  
let          2
```

- c. Now use the `assert` function to check the result:

```
const          '../checkFive.js'  
const          'assert'  
  
"checkFive"  function  
  
"Descriptive feedback..."  function  
let          2  
          "2 is less than 5."
```

(continues on next page)

(continued from previous page)

- d. Run the test script and examine the results. The test should pass and produce output similar to:

```
Started
.
.
1 spec, 0 failures
Finished in 0.006 seconds
```

- e. Now change line 3 in checkFive.js to if (num > 5) and rerun the test. The output should look similar to :

```
Started
F

Failures:
1) checkFive should return 'num' is less than 5 when passed a number smaller
   ↵than 5.

Message:
AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
+ expected - actual

- '2 is greater than 5.'
+ '2 is less than 5.'
```

- f. Change line 3 back.

Note

We do NOT need to check every possible value that is less than 5. Testing a single example is sufficient to check that part of the function.

4. Add two more `it` clauses inside `describe`—one to test what happens when `checkFive` is passed a value greater than 5, and the other to test when the value equals 5.

14.6.2 Try One on Your Own

Time for Rock, Paper, Scissors! The function below takes the choices ('rock', 'paper', or 'scissors') of two players as its parameters. It then decides which player won the match and returns a string.

```
1  function
2
3    if
4      return 'TIE!'
5
6
7    if          'rock'           'paper'
8      return 'Player 2 wins!'
9
10
11   if          'paper'         'scissors'
12     return 'Player 2 wins!'
```

(continues on next page)

(continued from previous page)

```
13
14
15     if           'scissors'          'rock '
16     return 'Player 2 wins!'
17
18
19     return 'Player 1 wins!'
20
```

repl.it

1. Set up the `RPS.js` and `RPS.spec.js` files to talk to each other. If you need to review how to do this, re-read the *previous exercise*, or check *Hello Jasmine*.
2. Write a test in `RPS.spec.js` to check if `whoWon` behaves correctly when the players tie (both choose the same option). Click “Run” and examine the output. SPOILER ALERT: The code for checking ties is correct in `whoWon`, so the test should pass. If it does not, modify your `it` statement.
3. Write tests (one at a time) for each of the remaining cases. Run the tests after each addition, and modify the code as needed. There is one mistake in `whoWon`. You might spot it on your own, but try to use automated testing to identify and fix it.

14.6.3 Bonus Mission

What if something OTHER than `'rock'`, `'paper'`, or `'scissors'` is passed into the `whoWon` function? Modify the code to deal with the possibility.

Don’t forget to add another `it` clause in `RPS.spec.js` to test for this case.

14.7 Studio: Unit Testing

Let’s use Test Driven Development (TDD) to help us design a function that meets the following conditions:

1. When passed a number that is ONLY divisible by 2, return `'Launch!'`
2. When passed a number that is ONLY divisible by 3, return `'Code!'`
3. When passed a number that is ONLY divisible by 5, return `'Rocks!'`
4. When passed a number that is divisible by 2 AND 3, return `'LaunchCode!'`
5. When passed a number that is divisible by 3 AND 5, return `'Code Rocks!'`
6. When passed a number that is divisible by 2 AND 5, return `'Launch Rocks!'`
7. When passed a number that is divisible by 2, 3, AND 5, return `'LaunchCode Rocks!'`
8. When passed a number that is NOT divisible by 2, 3, or 5, return `'Rutabagas! That doesn't work.'`

Rather than complete the code and *then* test it, TDD flips the process:

1. Write a test first - one that checks the program for a specific outcome.
2. Run the test to make sure it fails.
3. Write a code snippet that passes the test.
4. Repeat steps 1 - 3 for the remaining features of the program.

5. Examine the code and test scripts, and refactor them to increase efficiency. Remember the DRY idea (Don't Repeat Yourself).

14.7.1 Source Code

Open [this repl.it](#) and note the expected files:

1. `index.js` holds the Jasmine script. DO NOT CHANGE THIS FILE.
2. `launchCodeRocks.js` holds the function we want to design, which we will call `launchOutput`.
3. `launchCodeRocks.spec.js` holds the testing script.

Besides `index.js` the files are mostly empty. Only a framework has been provided for you.

14.7.2 Write the First Test

In `launchCodeRocks.spec.js`, complete the `describe` function by adding a test for condition #1:

When passed a number that is ONLY divisible by 2, `launchOutput` returns 'Launch!'

Run the test. It should fail because there is no code inside `launchOutput` yet!

14.7.3 Write Code to Pass the First Test

In `launchCodeRocks.js`, use an `if` statement inside the `launchOutput` function to check if the parameter is evenly divisible by 2, and then return an output. (*Hint: modulus*).

Run the test script again to see if your code passes. If not, modify `launchOutput` until it does.

14.7.4 Write the Next Two Tests

In `launchCodeRocks.spec.js`, add tests for the conditions:

2. When passed a number that is ONLY divisible by 3, `launchOutput` returns 'Code!'
3. When passed a number that is ONLY divisible by 5, `launchOutput` returns 'Rocks!'

Run the tests (you have three now). The two new ones should fail, but the first should still pass. Modify the `it` statements as needed if you see a different result.

14.7.5 Write Code to Pass the New Tests

Add more code inside `launchOutput` to check if the parameter is evenly divisible by 2, 3, or 5, and then return an output based on the result.

Run the test script again to see if your code passes all three tests. If not, modify `launchOutput` until it does.

14.7.6 Hmm, Tricky

In `launchCodeRocks.spec.js`, add a test for the condition:

4. When passed a number that is divisible by 2 AND 3, `launchOutput` returns 'LaunchCode!' (not 'Launch!Code!').

Run the tests. Only the new one should fail.

Modify `launchOutput` until the function passes all four of the tests.

14.7.7 More Tests and Code Snippets

Continue adding ONE test at a time for the remaining conditions. After you add EACH new test, run the script to make sure it FAILS, while the previous tests still pass.

Modify `launchOutput` until the function passes the new test and all of the old ones.

Presto! By starting with the *testing* script, you constructed `launchOutput` one segment at a time. The result is complete, valid code that has already been checked for accuracy.

14.7.8 New Condition

Now that your function passes all 8 tests, let's change one of the conditions. For the case where a number is divisible by both 2 and 5, instead of returning '`'Launch Rocks!'`', we want the function to return '`'Launch Rocks! (CRASH!!!!)'`'.

Modify the testing and function code to deal with this new condition.

14.7.9 Bonus Missions

DRYing the Code

Examine `launchOutput` and the `describe` functions. Notice that there is quite a bit of repetition in the code.

Try adding arrays, objects and/or loops to refactor the code into a more efficient structure.

What if We Already Have Code?

A teammate tried to help you out by writing the `launchOutput` code before class. Unfortunately, the code contains some errors.

Open the [flawed code here](#), and cut and paste your testing script into the `launchCodeRocks.spec.js` file.

Run the tests and see how many fail. Use the *descriptive feedback* from your `it` statements to find and fix the errors in `launchOutput`.

Note: You could just *cheat* and compare your correct code to the flawed sample, but the point of this mission is to give you more practice interpreting and using test results. To gain the most benefit, honor the spirit of this task.

SCOPE

15.1 Introduction

In the *Functions chapter*, we saw that *where* variables are declared and initialized in the code affects when they can be used. This idea is called **scope**, and it describes the ability of a program to access or modify a variable.

Example

```
1 let      0
2
3 function
4   let      2
5   return
6
```

a is accessible *inside* and *outside* of coolFunction().

b is only accessible *inside* of coolFunction().

Let's add some `console.log` statements to explore this code snippet.

Example

```
1 let      0
2
3
4 function
5   let      2
6     `a = ${ }, b = ${ }.`
7   return
8
9
10    1
11
12
13
14
```

Console Output

```
0
1
```

(continues on next page)

(continued from previous page)

```
a = 1, b = 2.  
ReferenceError: b is not defined
```

-
1. Lines 2 and 11 print the initial and incremented values of `a`.
 2. Line 13 calls `coolFunction()`, and line 6 prints the values of `a` and `b`. This shows that both variables are accessible within the function.
 3. Line 14 throws a `ReferenceError`, showing that `b` is not accessible outside of `coolFunction`.

15.1.1 Block/Local Scope

Local scope refers to variables declared and initialized inside a function or block. A *locally scoped* variable can only be referenced inside of the block or function where it is defined. In the example above, `b` has a local scope limited to `coolFunction()`. Referencing or attempting to update `b` outside of the function leads to a scoping error.

Try It!

The following code block has an error related to scope. Try to fix it!

```
1 function  
2     let      10  
3     return 10  
4  
5  
6
```

repl.it

15.1.2 Global Scope

Global scope refers to variables declared and initialized outside of a function and in the main body of the file. These variables are accessible to any function within a file. In the first example above, `a` has global scope.

Global scope is the default in JavaScript. If you assign a value to a variable WITHOUT first declaring it with `let` or `const`, then the variable automatically becomes global.

Example

```
1 // Code here CAN use newVariable.  
2  
3 function  
4     5  
5     return  
6  
7 // Code here CAN use newVariable.  
8
```

Warning

In the loop `for (i = 0; i < string.length; i++)`, leaving off the `let` from `i = 0` means that `i` is treated as a global variable. ANY other portion of the program can access or modify `i`, which could disrupt how well the loop operates.

15.1.3 Execution Context

Execution context refers to the conditions under which a variable is executed—its scope. Scoping affects the variable's behavior at runtime. When the code is run in the browser, everything is first run at a global context. As the compiler processes the code and finds a function, it shifts into the function context before returning to global execution context.

Let's consider this code:

```
1 let      0
2
3 function
4   let      0
5   return
6
7
8 function
9   let      0
10
11 return
12
```

Now, let's consider the execution context for each step.

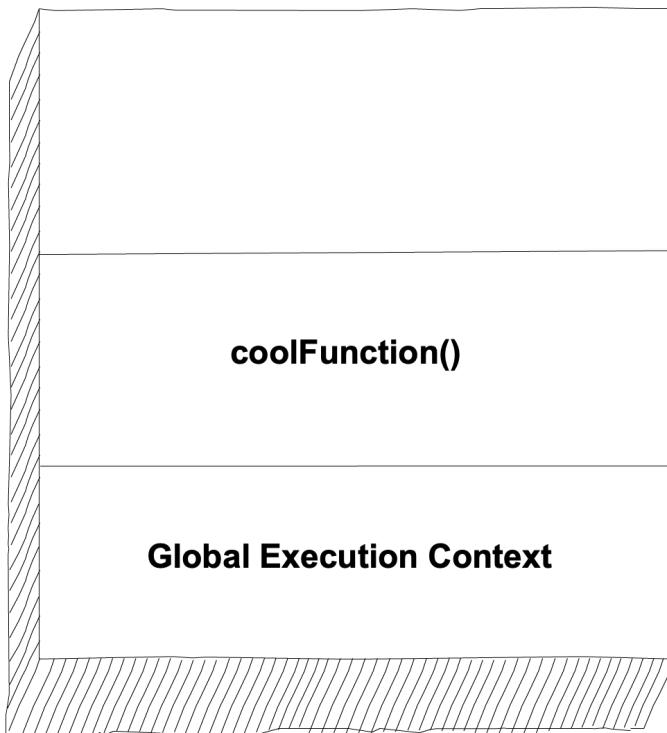
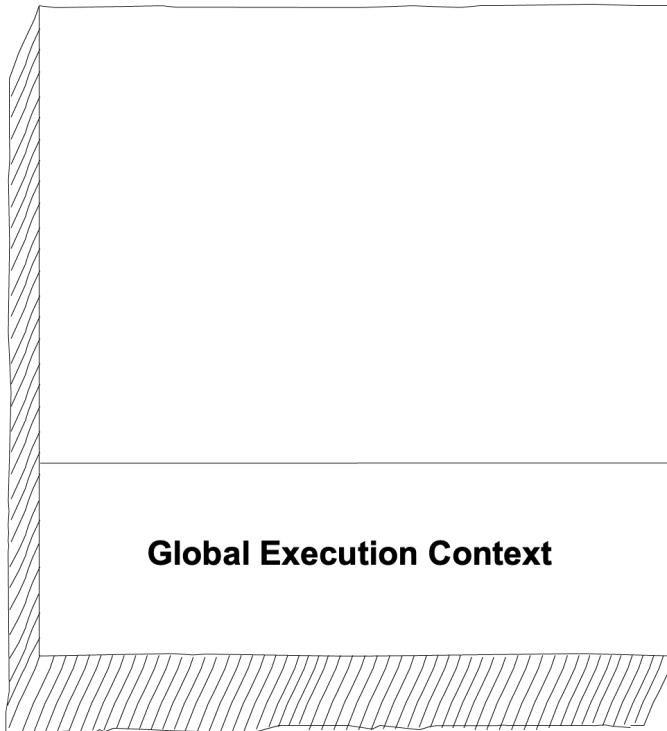
1. First, the global execution context is entered as the compiler executes the code.
2. Once `coolFunction()` is hit, the compiler creates and executes `coolFunction()` under the `coolFunction()` execution context.
3. Upon completion, the compiler returns to the global execution context.
4. The compiler stays at the global execution context until the creation and execution of `coolerFunction()`.
5. Inside of `coolerFunction()` is a call to `coolFunction()`. The compiler will go up in execution context to `coolFunction()` before returning down to `coolerFunction()`'s execution context. Upon completion of that function, the compiler returns to the global execution context.

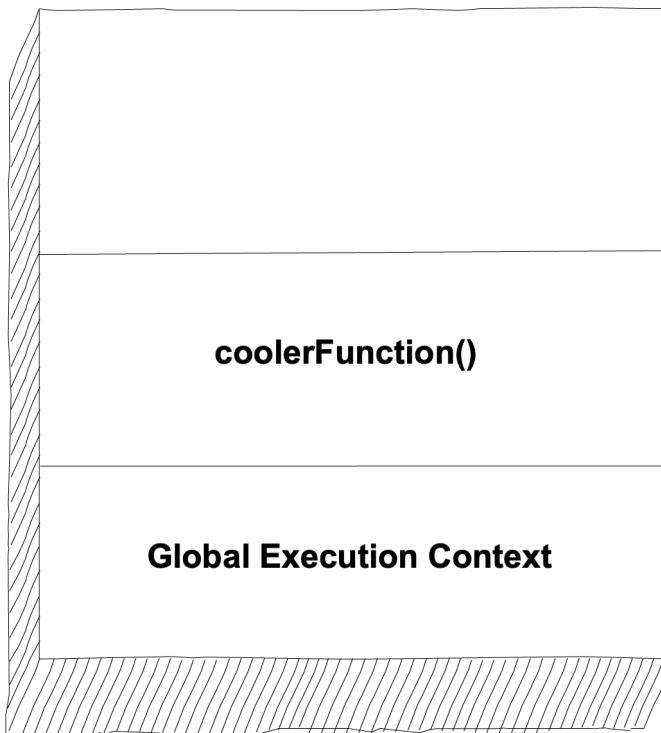
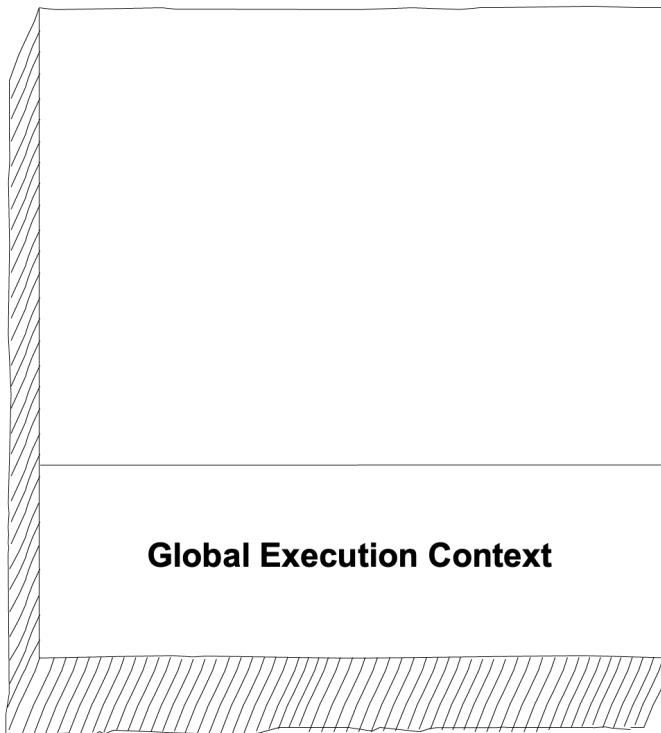
15.1.4 Check Your Understanding

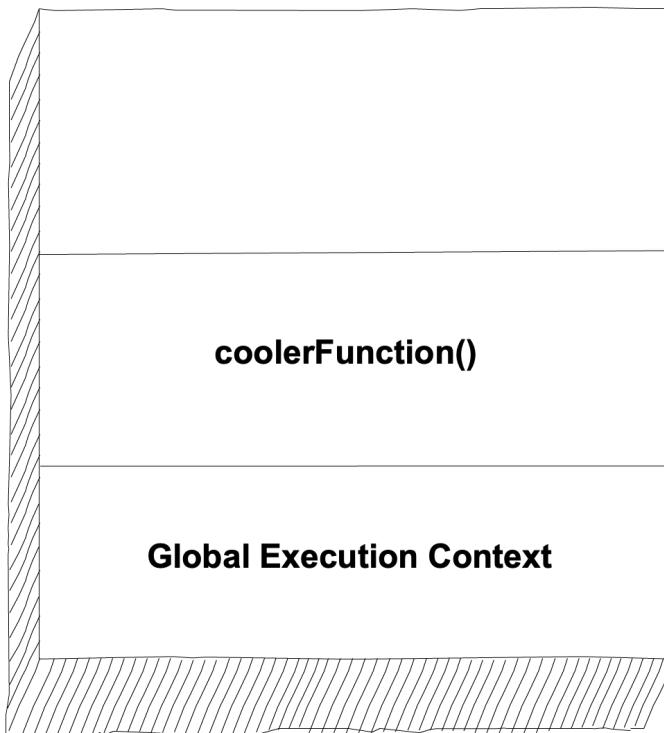
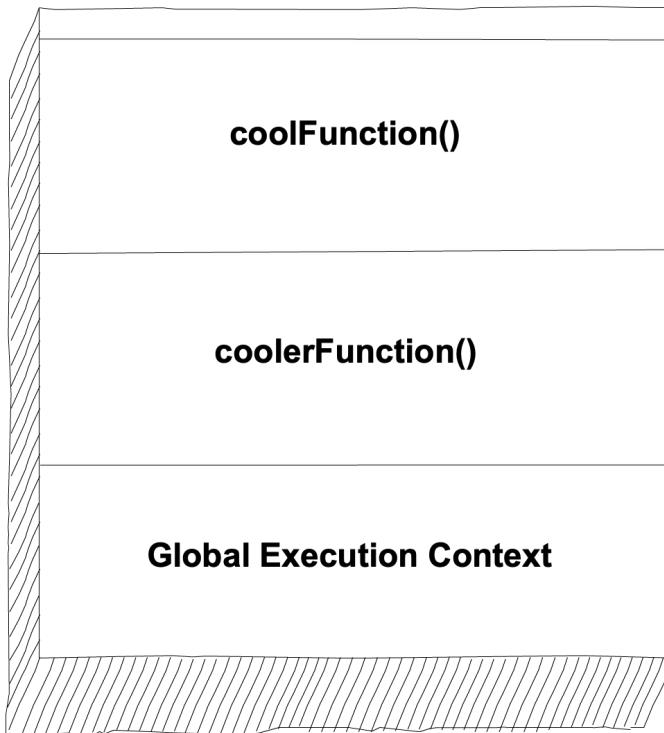
Both of the concept checks refer to the following code block:

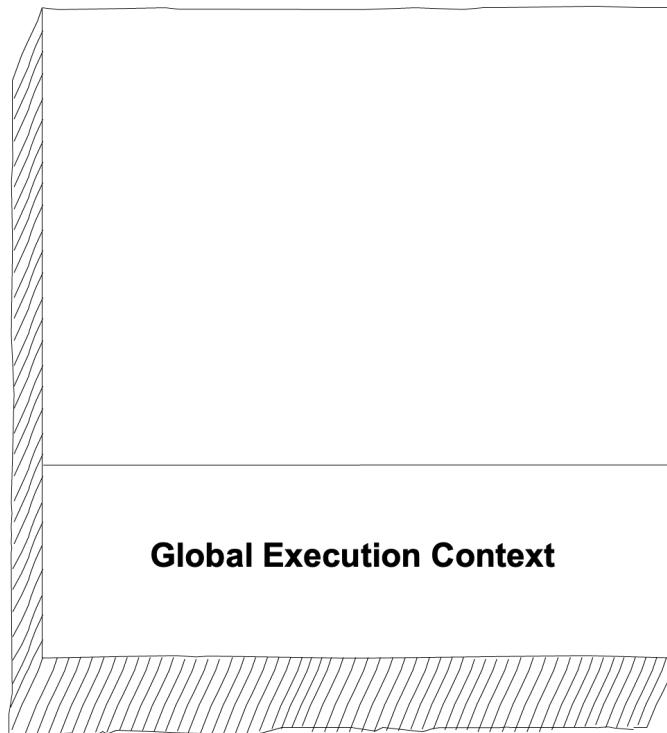
```
1 function
2   let      100
3   return
4
5
6 let      0
7
```

Question









What scope is variable `x`?

- a. Global
 - b. Local
-

Question

In what order will the compiler execute the code?

15.2 Using Scope

Scope allows programmers to control the flow of information through the variables in their program. Some variables you want to set as constants (like `pi`), which can be accessed globally. Others you want to keep secure to minimize the danger of accidental updates. For example, a variable holding someone's username should be kept secure.

15.2.1 Shadowing

Variable shadowing is where two variables in different scopes have the same name. The variables can then be accessed under different contexts. However, shadowing can affect the variable's accessibility. It also causes confusion for anyone reviewing the code.

Example

```
1 const           'readline-sync'
2
3 function
4     'Hello, '
5     'Ruth'
6 return
7
8
9 function
10
11 return
12
13
14 let           "Please enter your name: "
15
16
17
18
```

So, what is the value of name in line 4, 10, 16, 17, and 18?

Yikes! This is why shadowing is NOT a best practice in coding. Whenever possible, use different global and local variable names.

Try It!

If you are curious about the name values in the example, feel free to run the code [here](#).

15.2.2 Variable Hoisting

Variable hoisting is a behavior in JavaScript where variable declarations are raised to the top of the current scope. This results in a program being able to use a variable before it has been declared. Hoisting occurs when the `var` keyword is used in the declaration, but it does NOT occur when `let` and `const` are used in the declaration.

Note

Although we don't use the `var` keyword in this book, you will see it a lot in other JavaScript resources. Variable hoisting is an important concept to keep in mind as you work with JavaScript.

15.2.3 Check Your Understanding

Question

What keyword allows a variable to be hoisted?

- a. `let`
- b. `var`

c. const

Question

Consider this code:

```
1 let      0
2
3 function
4     let      10
5     return
6
```

Because there are two separate variables with the name, a, under the two different scopes, a is being shadowed.

- a. True
 - b. False
-

MORE ON TYPES

16.1 Primitive Data Types

In JavaScript, data types can fall into one of two categories: primitive or object types. A **primitive** data type is a basic building block. Using primitive data types, we can build more complex data structures or object data types.

While object types such as objects and arrays are mutable, primitive data types are immutable. Immutable data types are data types that cannot be changed once the value has been created.

Primitive data types include:

1. Strings
2. Numbers
3. Booleans
4. `undefined`
5. `null`

16.1.1 `undefined`

`undefined` is a primitive data type in JavaScript which is assigned to declared variables, which have *not* been initialized.

```
1 let
2
```

Console Output

```
undefined
```

16.1.2 `null`

`null` is similar to `undefined` in that it represents an unknown value, however, it is assigned to values that the programmer wishes to keep empty.

```
let null
```

Console Output

```
null
```

16.1.3 Example

Let's say that we are still working for the zoo. We have objects created for animals like so:

```
1 let
2     "Reticulated Giraffe"
3     "Cynthia"
4     1500
5     15
6     "leaves"
7
```

Now, a new giraffe is coming to the zoo. We may want to initialize an object for the giraffe, but hold off on storing information in the `weight` property until the giraffe arrives. In this case, we could initialize the `weight` property like so:

```
1 let
2     "Reticulated Giraffe"
3     "Alicia"
4     null
5     10
6     "leaves"
7
```

This way, our object is properly initialized with all of the information we would need and we can update the `weight` property later when we have accurate information.

16.1.4 Check Your Understanding

Question

Which of the following are primitive data types? Mark ALL that apply.

- a. arrays
 - b. Strings
 - c. objects
 - d. null
-

Question

Consider the following code block:

```
1 let
2
3
```

x is of what data type?

- a. null

- b. undefined
 - c. NaN
 - d. number
-

CHAPTER
SEVENTEEN

EXCEPTIONS

17.1 Introduction

Errors are a part of coding. Occasionally, we make mistakes as programmers. However, we are always trying to fix those mistakes by reading different resources, examining a list of error messages (also called the **stacktrace**), or asking for help.

Earlier in this course, we learned about two different types of errors: runtime and logic. A logic error is when your program executes without breaking, but doesn't behave the way you thought it would. These logic errors usually require you to consider how you are going about solving the issue to resolve. Runtime errors are when your program does not run correctly, and an exception is raised.

An **exception** is a runtime error in which a name and message are displayed to provide more information about the error.

17.1.1 Exceptions and Errors

In JavaScript a runtime error and an exception are the same thing and can be used interchangeably. This can cause confusion because a logic error is not an exception!

17.1.2 Error Object

When a runtime error, also known as an exception, is raised JavaScript returns an `Error` object. An Error Object has two properties: a name and a message. The name refers to the type of error that occurred, while the message gives the user information on why that exception occurred.

JavaScript has built-in exceptions with pre-defined names and messages, however, JavaScript also gives you the ability to create your own error messages.

You have undoubtedly experienced various Exceptions already throughout this class. Let's look at a few common Exceptions.

17.1.3 Common Exceptions

JavaScript has some built-in Exceptions you may have already encountered in this class.

One of the most common errors in Javascript is a `SyntaxError` which is thrown when we include a symbol JavaScript is not expecting.

Example

```
"This is"
```

Console Output

```
SyntaxError: missing ) after argument list
```

We put our second quotation mark in the incorrect place. JavaScript does not know what to do with the second half of our phrase and throws a `SyntaxError` with the message: `missing) after argument list`.

A `ReferenceError` is thrown when we try to use a variable that has not yet been defined.

Example

```
0
```

Console Output

```
ReferenceError: x is not defined
```

We attempt to print out the first element in the variable `x`, but we never declared `x`. JavaScript throws a `ReferenceError` with the message: `x is not defined`.

A `TypeError` is thrown when JavaScript expects something to be one type, but the provided value is a different type.

Example

```
1 const      "Launch"  
2  
3     "Code"
```

Console Output

```
TypeError: invalid assignment to const 'a'
```

In this case, we declare a constant as the string “Launch”, and then try to change the immutable variable to “Code”. JavaScript throws a `TypeError` with the message: `invalid assignment to const 'b'`.

Exceptions give us a way to provide more information on how something went wrong. JavaScript’s built-in Exceptions are regularly used in the debugging process.

There are more built-in Exceptions in Java, you can read more by referencing the [MDN Errors Documentation](#) or [W3Schools JavaScript Error](#) (scroll down to the Error Object section).

In the next section we will learn how to raise our own exceptions using the `throw` statement.

17.1.4 Check Your Understanding

Question

What is the difference between a runtime error, and a logic error?

Question

What are some of the common errors included in JavaScript?

17.2 Throw

In most programming languages, when the compiler or interpreter encounters code it doesn't know how to handle, it **throws** an exception. This is how the compiler notifies the programmer that something has gone wrong. Throwing an exception is also known as *raising* an exception.

JavaScript gives us the ability to raise exceptions using the `throw` statement. One reason to throw an exception is if your code is being used in an unexpected way.

17.2.1 Throw Default Error

We can throw a default Error by using the `throw` statement and passing in a string description as a argument.

Example

```
throw      "You cannot divide by zero!"
```

repl.it

Console Output

```
Error: You cannot divide by zero!
at evalmachine.<anonymous>:1:7
at Script.runInContext (vm.js:133:20)
at Object.runInContext (vm.js:311:6)
at evaluate (/run_dir/repl.js:133:14)
```

The error text displays the error name, and it contains details about where the error was thrown. The text at `evalmachine.<anonymous>:1:7` indicates that the error was thrown from line 1, which we know is true because our example only has one line of code.

17.2.2 Pre-existing Error

JavaScript also gives us the power to throw a more specific type of error.

Example

```
throw      "That is the incorrect syntax"
```

Console Output

```
SyntaxError: That is the incorrect syntax
```

JavaScript gives us flexibility by allowing us to raise standard library errors and to define our own errors. We can use exceptions to allow our program to break and provide useful information as to why something went wrong.

17.2.3 Custom Error

JavaScript will also let you define new types of Errors. You may find this helpful in the future, however, that goes beyond the scope of this class.

17.2.4 Check Your Understanding

Question

What statement do we use to raise an exception?

Question

How do we customize the message of an exception?

17.3 Exceptions as Control Flow

Runtime errors occur as the program runs, and they are also called exceptions. Exceptions are caused by referencing undeclared variables and invalid or unexpected data.

17.3.1 Control Flow

The **control flow** of a program is the order in which the statements are executed. Normal control flow runs from top to bottom of a file. An exception breaks the normal flow and stops the program. A stopped program can no longer interact with the user. Luckily JavaScript provides a way to anticipate and handle exceptions.

17.3.2 Catching an Exception

JavaScript provides `try` and `catch` statements that allow us to keep our programs running even if there is an exception. We can tell JavaScript to *try* to run a block of code, and if an exception is thrown, to *catch* the exception and run a specific block of code. Anticipating and catching the exception makes the exception now part of the control flow.

Note

Catching an exception is also known as *handling* an exception.

Example

In this example there is an array of animals. The user is asked to enter the index for the animal they want to see. If the user enters an index that does NOT contain an animal, the code will throw a `TypeError` when `name` is referenced on an undefined value.

There is a `try` block around the code that will throw the `TypeError`. There is a `catch` block that catches the error and contains code to inform the user that they entered an invalid index.

```

1 const           'readline-sync'
2
3 let            'cat'        'dog'
4 let            "Enter index of animal:"
5
6 try           'animal at index:'
7
8 catch         "We caught a TypeError, but our program continues to run!"
9             "You tried to access an animal at index:"
10
11
12
13 "the code goes on..."

```

repl.it

Console Output

If the user enters 9:

```

Enter index of animal: 9
We caught a TypeError, but our program continues to run!
You tried to access an animal at index: 9
the code goes on...

```

If the user enters 0:

```

Enter index of animal: 0
animal at index: cat
the code goes on...

```

Tip

`catch` blocks only execute if an exception is thrown

17.3.3 Finally

JavaScript also provides a `finally` block which can be used with `try` and `catch` blocks. A `finally` block code runs after the `try` and `catch`. What is special about `finally` is that `finally` code block ALWAYS runs, even if an exception is NOT thrown.

Example

Let's update the above example to print out the index the user entered. We want this message to be printed EVERY time the code runs. Notice the `console.log` statement on line 11.

```

1 const           'readline-sync'
2
3 let            'cat'        'dog'
4 let            "Enter index of animal:"
5
6 try           'animal at index:'
7
8 catch         "We caught a TypeError, but our program continues to run!"
9             "You tried to access an animal at index:"
10
11 console.log("User entered index: " + index)
12
13 "the code goes on..."

```

(continues on next page)

(continued from previous page)

```
8   catch
9       "We caught a TypeError, but our program continues to run!"
10  finally
11      "You tried to access an animal at index:"
12
13
14      "the code goes on..."
```

repl.it

Console Output

If the user enters 7:

```
Enter index of animal: 7
We caught a TypeError, but our program continues to run!
You tried to access an animal at index: 7
the code goes on...
```

If the user enters 1:

```
Enter index of animal: 1
animal at index: dog
You tried to access an animal at index: 1
the code goes on...
```

17.3.4 Check Your Understanding

Question

What statement do we use if we want to attempt to run code, but think an exception might be thrown?

1. catch
 2. try
 3. throw
 4. finally
-

Question

How do you handle an exception that is thrown?

1. With code placed within the `try` block.
 2. With code placed within the `catch` block.
 3. With code placed within a `throw` statement.
 4. With code placed within the `finally` block.
-

Question

What statement do you use to ensure a code block is executed regardless if an exception was thrown?

1. throw
 2. catch
 3. try
 4. finally
-

17.4 Exercises

17.4.1 Zero Division: Throw

Write a function called divide that takes two parameters: a numerator and a denominator.

Your function should return the result of numerator / denominator.

However, if the denominator is zero you should throw an error informing the user they attempted to divide by zero. The error text should be Attempted to divide by zero.

Note

Hint: You can use an if / throw statement to complete this exercise.

17.4.2 Test Student Labs

A teacher has created a gradeLabs function that verifies if student programming labs work. This function loops over an array of JavaScript objects that *should* contain a student property and runLab property.

The runLab property is expected to be a function containing the student's code. The runLab function is called and the result is compared to the expected result. If the result and expected result don't match, then the lab is considered a failure.

```
1  function
2    for let 0
3      let
4        let
5          `${ }` code worked: ${ 27 }`  
6
7
8
9 let
10   'Carly'
11   function
12     return
13
14
15
16   'Erica'
17   function
18     return
19
20
21
```

(continues on next page)

(continued from previous page)

22
23
24

The gradeLabs function works for the majority of cases. However what happens if a student named their function incorrectly? Run gradeLabs and pass it studentLabs2 as defined below.

```
1 let
2
3     'Jim'
4         function
5             return
6
7
8
9     'Jessica'
10        function
11        return
12
13
14
15
16
```

Upon running the second example, the teacher gets a `TypeError: lab.runLab is not a function`.

Your task is to add a `try catch` block inside of `gradeLabs` to catch an exception if the `runLab` property is not defined.

17.5 Studio: Strategic Debugging

17.5.1 Summary

At this point, we have seen a lot of different types of errors. We have possibly created logic errors or syntax errors and now, we have just learned about the `Error` object in JavaScript. The goal of this studio is for us to develop strategies for debugging so that we can get rid of the bugs and get back to coding!

17.5.2 Activity

Think of a bug you have seen in your code. This could be the time you dropped a keyword when initializing a variable or misused a method.

1. Take some time to discuss with the group what your error was and how you solved it. Did you talk to a TA to get it? Did you find a great resource online that was helpful?

Your TA will go over the pros and cons of different resources that can help you resolve the error. You will then go over a general strategy to start debugging your errors.

17.5.3 Debugging Process

Your TA will go over this process with you and how it could help you debug more strategically. This process reflects what we have found works best for us and many students, however, as you grow as a programmer, you may find

something works better for you. That is fine! Every programmer has their own process for debugging based off of their experiences and how their mind works.

1. Check the *stacktrace* to read the error message and see where it occurred.
2. If you see the error, fix it on that line and recompile.
3. If you cannot see the error, Google the error message.
4. Check any relevant StackOverflow posts in the results.
5. If the error is related to built-in methods or objects, also search for those in the official documentation.
6. If the error is related to something that cannot be done in that particular language, look at the responses to each comment before trying to replicate proposed solutions. Solutions can oftentimes go out of date and responses will tell you if that is the case or simply if it is a bad solution.

CHAPTER
EIGHTEEN

CLASSES

18.1 What Are Classes?

Recall that *objects* are data structures that hold many values, which consist of *properties* and *methods*.

We often need to create many objects of the same *type*. To do this in an efficient way, we define a **class**, which allows us to set up the general structure for an object. We can then reuse that structure to build multiple objects. These objects all have the same set of *keys*, but the *values* assigned to each key will vary.

Let's revisit the animal astronauts from earlier exercises to see how this works.

18.1.1 An Astronaut Object

When we create an object to hold an astronaut's data, it might look something like:

```
1 let
2   'Fox'
3   7
4   12
5     function
6   let      'LaunchCode'
7   for let    0
8     'Rocks'
9
10 return
11
12
13
14   `${      } is ${      } years old and has a mass of ${      } kg.`
15   `${      } says, "${      } .".`
```

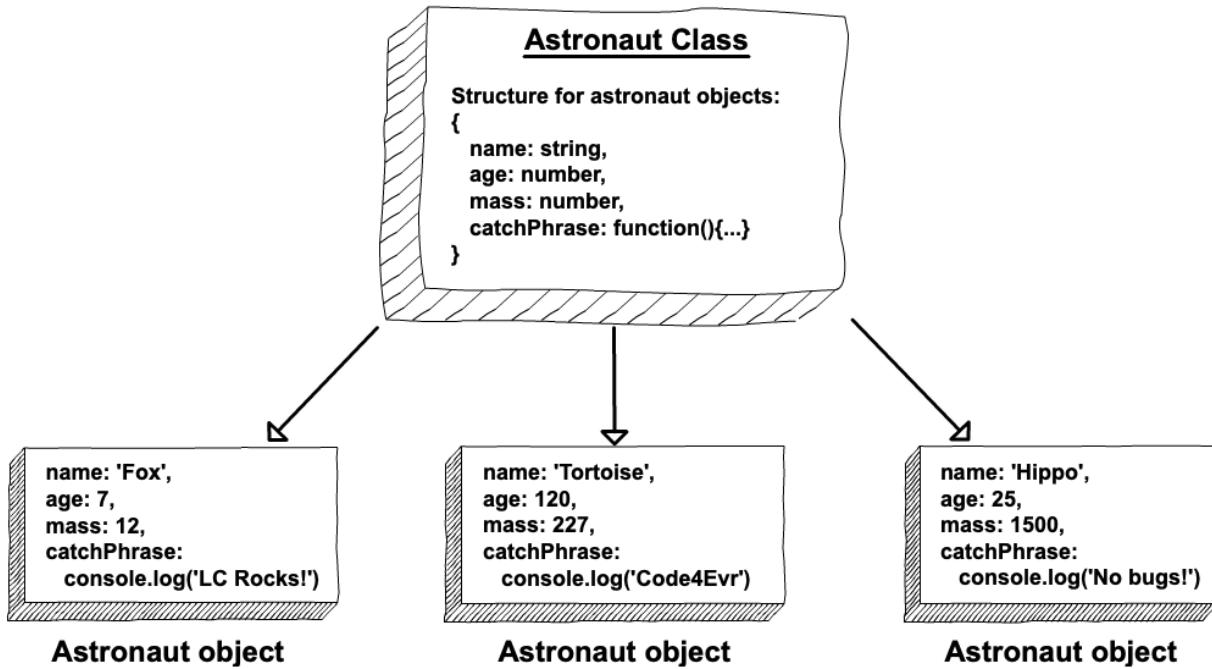
Console Output

```
Fox is 7 years old and has a mass of 12 kg.
Fox says, "LaunchCode Rocks Rocks Rocks."
```

The `fox` object contains all the data and functions for the astronaut named '`Fox`'.

Of course, we have multiple astronauts on our team. To store data for each one, we would need to copy the structure for `fox` multiple times and then change the values to suit each crew member. This is inefficient and repetitive.

By letting us define our own **classes**, JavaScript provides a better way to create multiple, similar objects.



18.2 Declaring and Calling a Class

18.2.1 Creating a Class

Just like the `function` keyword defines a new function, the keyword for defining a new class is `class`. By convention, class names start with capital letters to distinguish them from JavaScript function and variable names (e.g. `MyClass` vs. `myFunction`).

Remember that classes are blueprints for building multiple objects of the same type. The general format for declaring a class is:

```
1 class
2
3   //assign properties
4
5   //define methods
6
```

Note the keyword `constructor`. This is a special method for creating objects of the same type, and it assigns the key/value pairs. Parameters are passed into `constructor` rather than the `class` declaration.

Assigning Properties

Let's set up an `Astronaut` class to help us store data about our animal crew. Each animal has a name, age, and mass, and we assign these properties in `constructor` as follows:

```
1 class
2
3   this
4   this
```

(continues on next page)

(continued from previous page)

```
5      this  
6  
7
```

The `this` keyword defines a key/value pair, where the text attached to `this` becomes the key, and the value follows the equal sign (`this.key = value`).

`constructor` uses the three `this` statements (`this.name = name`, etc.) to achieve the same result as the object declaration `let objectName = {name: someString, age: someNumber, mass: someMass}`. Each time the `Astronaut` class is called, `constructor` builds an object with the SAME set of keys, but it assigns different values to the keys based on the arguments.

Note

Each class requires *one* `constructor`. Including more than one `constructor` results in a syntax error. If `constructor` is left out of a class declaration, JavaScript adds an empty `constructor () {}` automatically.

18.2.2 Creating a New Class Object

To create an object from a class, we use the keyword `new`. The syntax is:

```
let      new
```

`new` creates an **instance** of the class, which means that the object generated shares the same set of keys as every other object made from the class. However, the values assigned to each key differ.

For this reason, objects created with the same class are NOT equal.

Example

Let's create objects for two of our crew members: Fox and Hippo.

```
1  class  
2  
3      this  
4      this  
5      this  
6  
7  
8  
9  let      new      'Fox'  7  12  
10 let      new      'Hippo' 25  1500  
11  
12      typeof      typeof  
13  
14
```

Console Output

```
object object  
  
Astronaut { name: 'Hippo', age: 25, mass: 1500 }  
Astronaut { name: 'Fox', age: 7, mass: 12 }
```

In lines 9 and 10, we call the `Astronaut` class twice and pass in different sets of arguments, creating the `fox` and `hippo` objects.

The output of line 14 shows that `fox` and `hippo` are both the same *type* of object (`Astronaut`). The two share the same *keys*, but they have different values assigned to those keys.

After creating an `Astronaut` object, we can access, modify, or add new key/value pairs as described in the *Objects and Math chapter*.

Try It

Play around with modifying and adding properties inside and outside of the `class` declaration.

```
1  class
2
3      this
4      this
5      this
6
7
8
9  let      new      'Fox'  7  12
10
11
12
13
14      9
15      'red'
16
17
18
```

repl.it

Console Output

```
Astronaut { name: 'Fox', age: 7, mass: 12 }
7 undefined
Astronaut { name: 'Fox', age: 9, mass: 12, color: 'red' }
9 'red'
```

Attempting to print `fox.color` in line 12 returns `undefined`, since that property is not included in the `Astronaut` class. Line 15 adds the `color` property to the `fox` object, but this change will not affect any other objects created with `Astronaut`.

Setting Default Values

What happens if we create a new `Astronaut` without passing in all of the required arguments?

Try It!

```
1  class
2
3      this
```

(continues on next page)

(continued from previous page)

```
4      this  
5      this  
6  
7  
8  
9 let           new          'Speedy'  120  
10  
11
```

repl.it

To avoid issues with missing arguments, we can set a *default* value for a parameter as follows:

```
1 class  
2             54  
3     this  
4     this  
5     this  
6  
7
```

Now if we call `Astronaut` but do not specify a mass value, the constructor automatically assigns a value of 54. If an argument is included for `mass`, then the default value is ignored.

TRY IT! Return to the repl.it in the example above and set default values for one or more of the parameters.

18.2.3 Check Your Understanding

The questions below refer to a class called `Car`.

```
1 class  
2  
3     this  
4     this  
5     this  
6     this  
7     this  
8  
9
```

Question

If we call the class with `let myCar = new Car('Chevy', 'Astro', 1985, 'gray', 20)`, what is output by `console.log(typeof myCar.year)`?

- a. object
 - b. string
 - c. function
 - d. number
 - e. property
-

Question

If we call the class with `let myCar = new Car('Tesla', 'Model S', 2019)`, what is output by `console.log(myCar)`?

- a. `Car {make: 'Tesla', model: 'Model S', year: 2019, color: undefined, mpg: undefined }`
 - b. `Car {make: 'Tesla', model: 'Model S', year: 2019, color: "", mpg: "" }`
 - c. `Car {make: 'Tesla', model: 'Model S', year: 2019 }`
-

18.3 Assigning Class Methods

Just as with objects, we may want to add methods to our classes in addition to properties. So far, we have learned how to set the values of the class's properties inside the `constructor`.

When assigning methods in classes, we can either create them *outside* or *inside* the `constructor`.

18.3.1 Assigning Methods Outside constructor

When assigning methods outside of the `constructor`, we simply declare our methods the same way we would normally do for *objects*.

```
1 class
2
3     //assign properties with this.key = value
4
5
6     //function code
7
8
9
10
```

Example

```
1 class
2
3     this
4     this
5     this
6
7
8
9     let      `${this}      } is ${this}      } years old and has a mass of ${this}
10    ↵      } kg.`
11     return
12
13
14 let      new      'Fox'  7  12
15
```

Console Output

```
Fox is 7 years old and has a mass of 12 kg.
```

We declared our method, `reportStats()` outside of the constructor. When we declare a new instance of the `Astronaut` class, we can use the `reportStats()` method to return a concise string containing all of the info we would need about an astronaut.

18.3.2 Assigning Methods Inside constructor

When declaring methods inside the `constructor`, we need to make use of the `this` keyword, just as we would with our properties.

```
1 class
2
3     this          function
4         //function code
5
6
7
```

Example

Let's consider the `Astronaut` class that we have been working with. When assigning the method, `reportStats()`, inside the `constructor`, we would do so like this:

```
1 class
2
3     this
4     this
5     this
6     this          function
7         let      `${this}      } is ${this}      } years old and has a mass of ${this}
8         } kg.
9             return
10
11
12 let      new      'Fox'  7  12
13
14
15
```

Console Output

```
Fox is 7 years old and has a mass of 12 kg.
```

Initially, this may seem to produce the same result as assigning `reportStats()` outside of the `constructor`. We will weigh the pros and cons of both methods below.

18.3.3 Which Way is Preferred?

Try It!

Try comparing the outputs of `fox` and `hippo` to see the effect of assigning a method *inside* the constructor versus *outside* the constructor.

```
1 // Here we assign the method inside the constructor
2 class
3
4     this
5     this
6     this
7     this           function
8         let      `${this}      } is ${this}      } years old and has a mass of ${this
9             } kg.`
10            return
11
12
13
14 // Here we assign the method outside fo the constructor
15 class
16
17     this
18     this
19     this
20
21
22
23     let      `${this}      } is ${this}      } years old and has a mass of ${this
24             } kg.`
25            return
26
27
28 let      new      'Fox'  7  12
29 let      new      'Hippo' 25  1000
30
31
32
```

repl.it

In the case of assigning the method *inside* the constructor, each `Astronaut` object carries around the code for `reportStats()`. With today's computers, this is a relatively minor concern. However, each `Astronaut` has extra code that may not be needed. This consumes memory, which you need to consider since today's businesses want efficient code that does not tax their systems.

Because of this, if a method is the same for ALL objects of a class, define that method *outside* of the constructor. Each object does not need a copy of identical code. Therefore, the declaration of a method outside of the constructor will not consume as much memory.

18.3.4 Check Your Understanding

Question

What is the assignment for the `grow` method missing?

```
1 class
2
3     this
4     this
5
6
7     this      this      1
8
9
10
```

18.4 Inheritance

Object-oriented programming is a type of software design where the codebase is organized around *objects* and *classes*. Objects contain the functions and central logic of a program.

Object-oriented programming stands on top of four principles: abstraction, polymorphism, encapsulation, and inheritance. We will dive into inheritance now and work with the other three principles in Unit Two of this class.

Inheritance refers to the ability of one class to acquire properties and methods from another.

Think of it this way, in the animal kingdom, a *species* is a unique entity that inherits traits from its *genus*. The *genus* also has unique properties, but inherits traits from its *family*. For example, a tiger and a housecat are members of two different species, however, they share similar traits such as retractable claws. The two cats inherited their similar traits from their shared family, *felidae*.

Using inheritance in programming, we can create a structure of classes that inherit properties and methods from other classes.

If we wanted to program classes for our tiger and housecat, we would create a *felidae* class for the family. We would then create two classes for the *panthera* genus and the *felis* genus. We would create classes for the tiger and house cat species as well. The species classes would inherit properties and methods from the genus classes and the genus classes would inherit properties and methods from the family class.

The classes inheriting properties and methods are **child classes**, and the classes passing down properties and methods are **parent classes**.

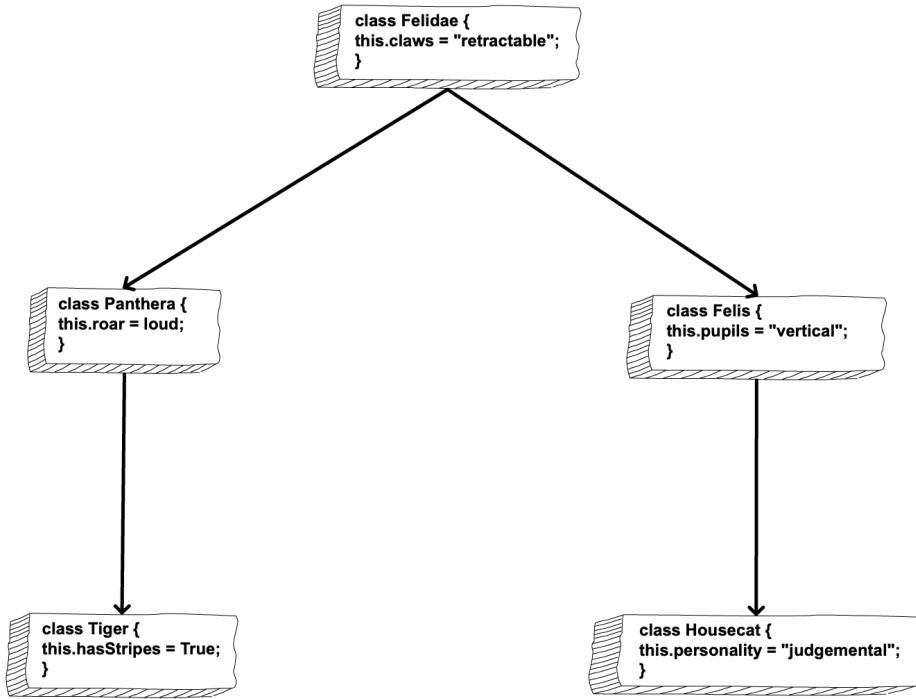
18.4.1 extends

When designating a class as the child class of another in JavaScript, we use the `extends` keyword. We also must use the `super ()` constructor to get the properties and methods needed from the parent class.

```
1 class      extends
2
3     super
4     // properties
5
6
```

In the case of a tiger, tigers have stripes, but they also have loud roars. Their ability to roar loudly is a trait they share with other members of the *panthera* genus. Tigers also got their retractable claws from the *felidae* family.

Example



```

1  class
2
3      this      "retractable"
4
5
6
7  class      extends
8
9      super
10     this      "loud"
11
12
13
14  class      extends
15
16      super
17     this      "true"
18
19
20
21  let      new
22
23
  
```

[repl.it](#)

When creating the classes for our tiger, we can use the `extends` keyword to set up `Tiger` as the child class of `Panthera`. The `Tiger` class then inherits the property, `roar`, from the `Panthera` class and has an additional

property, `hasStripes`.

Note

The `extends` keyword is not supported in Internet Explorer.

18.4.2 Check Your Understanding

Question

If you had to create classes for a *wolf*, the *canis* genus, and the *carnivora* order, which statement is TRUE about the order of inheritance?

- a. Wolf and Canis are parent classes to Carnivora.
 - b. Wolf is a child class of Canis and a parent class to Carnivora.
 - c. Wolf is child class of Canis, and Canis is a child class of Carnivora.
 - d. Wolf is child class of Canis, and Canis is a parent class of Carnivora.
-

18.5 Exercises: Classes

Welcome to the space station! It is your first day onboard and as the newest and most junior member of the crew, you have been asked to organize the library of manuals and fun novels for the crew to read. Click on this [repl.it link](#) and fork the starter code.

Headquarters have asked that you store the following information about each book.

1. The title
2. The author
3. The copyright date
4. The ISBN
5. The number of pages
6. The number of times the book has been checked out.
7. Whether the book has been discarded.

Headquarters also needs you to track certain actions that you must perform when books get out of date. First, for a manual, the book must be thrown out if it is over 5 years old. Second, for a novel, the book should be thrown out if it has been checked out over 100 times.

1. Construct three classes that hold the information needed by headquarters as properties. Also, each class needs two methods that update the book's property if the book needs to be discarded. One class should be a `Book` class and two child classes of the `Book` class called `Manual` and `Novel`. *Hint:* This means you need to read through the requirements for the problem and decide what should belong to `Book` and what should belong to the `Novel` and `Manual` classes.
2. Declare an object of the `Novel` class for the following tome from the library:

Table 1: Novel

Variable	Value
Title	Pride and Prejudice
Author	Jane Austen
Copyright date	1813
ISBN	11111111111111
Number of pages	432
Number of times the book has been checked out	32
Whether the book has been discarded	No

3. Declare an object of the `Manual` class for the following tome from the library:

Table 2: Manual

Variable	Value
Title	Top Secret Shuttle Building Manual
Author	Redacted
Copyright date	2013
ISBN	00000000000000
Number of pages	1147
Number of times the book has been checked out	1
Whether the book has been discarded	No

4. One of the above books needs to be discarded. Call the appropriate method for that book to update the property. That way the crew can throw it into empty space to become debris.
5. The other book has been checked out 5 times since you first created the object. Call the appropriate method to update the number of times the book has been checked out.

18.6 Studio: Classes

Let's create a class to handle new animal crew candidates!

Edit the [practice file](#) as you complete the studio activity.

18.6.1 Part 1 - Add Class Properties

1. Declare a class called `CrewCandidate` with a constructor that takes three parameters—name, mass, and scores. Note that scores will be an array of test results.
2. Create objects for the following candidates:
 - a. Bubba Bear has a mass of 135 kg and test scores of 88, 85, and 90.
 - b. Merry Maltese has a mass of 1.5 kg and test scores of 93, 88, and 97.
 - c. Glad Gator has a mass of 225 kg and test scores of 75, 78, and 62.

Use `console.log` for each object to verify that your class correctly assigns the key/value pairs.

18.6.2 Part 2 - Add First Class Method

As our candidates complete more tests, we need to be able to add the new scores to their records.

1. Create an `addScore` method in `CrewCandidate`. The function must take a new score as a parameter. Code this function OUTSIDE of `constructor`. (If you need to review the syntax, revisit *Assigning Class Methods*).
2. When passed a score, the function adds the value to `this.scores` with the *push array method*.
3. Test out your new method by adding a score of 83 to Bubba's record, then print out the new score array with `objectName.scores`.

18.6.3 Part 3 - Add More Methods

Now that we can add scores to our candidates' records, we need to be able to evaluate their fitness for our astronaut program. Let's add two more methods to `CrewCandidate`—one to average the test scores and the other to indicate if the candidate should be admitted.

Calculating the Test Average

1. Add an `average()` method outside `constructor`. The function does NOT need a parameter.
2. To find the average, add up the entries from `this.scores`, then divide the sum by the number of scores.
3. To make the average easier to look at, *round it to 1 decimal place*, then return the result from the function.

Verify your code by evaluating and printing Merry's average test score (92.7).

Determining Candidate Status

Candidates with averages at or above 90% are automatically accepted to our training program. Reserve candidates average between 80 - 89%, while probationary candidates average between 70 - 79%. Averages below 70% lead to a rejection notice.

1. Add a `status()` method to `CrewCandidate`. The method returns a string (`Accepted`, `Reserve`, `Probationary`, or `Rejected`) depending on a candidate's average.
2. The `status` method requires the average test score, which can be called as a parameter OR from inside the function. That's correct - methods can call other methods inside a class! Just remember to use the `this` keyword.
3. Once `status` has a candidate's average score, evaluate that score, and return the appropriate string.
4. Test the `status` method on each of the three candidates. Use a template literal to print out ' earned an average test score of % and has a status of .'

18.6.4 Part 4 - Play a Bit

Use the three methods to boost Glad Gator's status to `Reserve` or higher. How many tests will it take to reach `Reserve` status? How many to reach `Accepted`? Remember, scores cannot exceed 100%.

Tip

Rather than adding one score at a time, could you use a loop?

TERMINAL

19.1 What is a terminal?

19.1.1 GUIs and CLIs

Most of the time when we use our computers, we do so through a **graphical user interface**, or **GUI** for short. A GUI is a system designed with icons and visual representations of the machine's file systems.

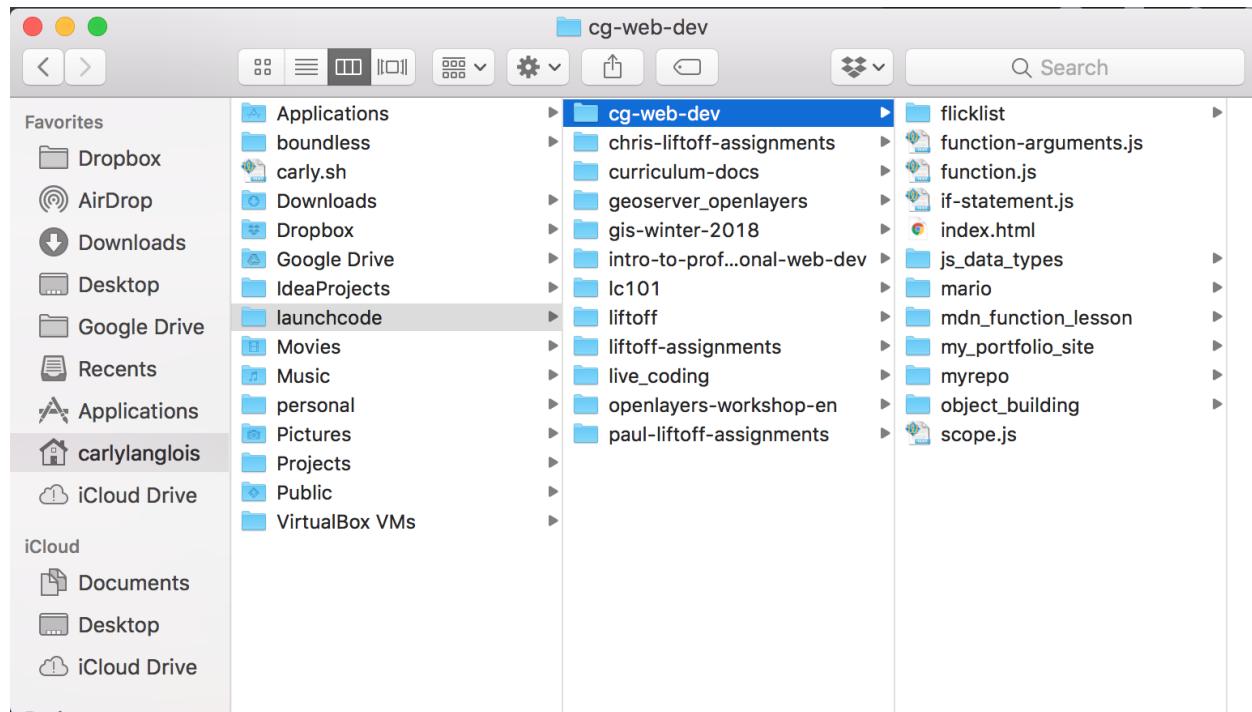
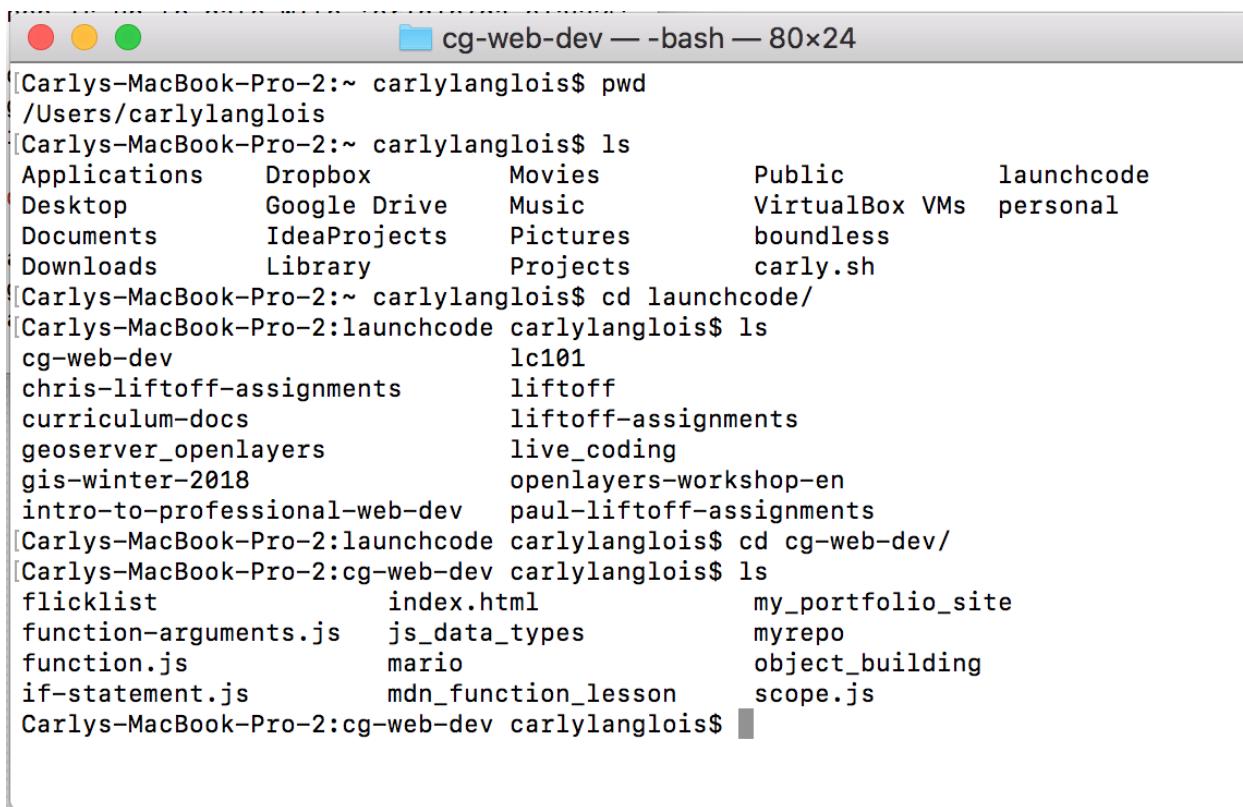


Fig. 1: A GUI with file icons and columns representing folder structure.

Programmers often use another kind of interface, called the **command line**. A **CLI**, or command line interface, uses textual commands, rather than dragging and dropping icons, to give the computer instructions.

The application responsible for running a CLI is called a **terminal** and the program interpreting the commands is called the **shell**.

Note



```
[Carlys-MacBook-Pro-2:~ carlylanglois$ pwd
'/Users/carlylanglois'
[Carlys-MacBook-Pro-2:~ carlylanglois$ ls
Applications      Dropbox      Movies          Public        launchcode
Desktop           Google Drive   Music           VirtualBox VMs  personal
Documents         IdeaProjects  Pictures        boundless
Downloads         Library       Projects       carly.sh
[Carlys-MacBook-Pro-2:~ carlylanglois$ cd launchcode/
[Carlys-MacBook-Pro-2:launchcode carlylanglois$ ls
cg-web-dev          lc101
chris-liftoff-assignments  liftoff
curriculum-docs      liftoff-assignments
geoserver_openlayers  live_coding
gis-winter-2018       openlayers-workshop-en
intro-to-professional-web-dev paul-liftoff-assignments
[Carlys-MacBook-Pro-2:launchcode carlylanglois$ cd cg-web-dev/
[Carlys-MacBook-Pro-2:cg-web-dev carlylanglois$ ls
flicklist            index.html        my_portfolio_site
function-arguments.js js_data_types    myrepo
function.js           mario           object_building
if-statement.js       mdn_function_lesson scope.js
Carlys-MacBook-Pro-2:cg-web-dev carlylanglois$ ]]
```

Fig. 2: A CLI with commands navigating the same file paths as the GUI above.

The terms “command line”, “terminal”, and “shell” are often used interchangeably.

19.1.2 Why use the terminal?

Both of the images above represent the same file structure. While the GUI may now appear more user-friendly, as you grow more familiar with the commands available, you’ll find there can be advantages to using the terminal.

In the terminal, you will be able to:

- quickly move throughout your computer’s file structure
- make new files and directories
- remove items from folders
- install software
- open programs
- run programs directly

19.2 Filesystem and Paths

A **filesystem** is a structure for the computer to store the files and folders that make up the data of the operating system.

Inside a filesystem, folders are referred to as **directories**. Your **root directory** is your Home folder or C drive. When you open a new terminal window to work in, it opens to your root directory. The root directory is the **parent directory** for the folders stored inside of it.

Example

Oftentimes, there is a Desktop folder inside the root directory. If there is a folder on your Desktop called “LC101_Homework”, then the parent directory of LC101_Homework is Desktop. The parent directory of Desktop is LC101_Homework.

A **path** for files and folders is the list of parent directories that the computer must go through to find that particular item.

Computers have two different types of paths: absolute and relative. The **absolute path** is the path to a file from the root directory. The **relative path** is the path to a file from the current directory. When working with a relative path, you may find yourself wanting to go up into a parent directory to find a file in a different child directory. In order to do so, you can use `..` in the file path to tell the computer to go up to the parent directory.

Example

We have a file inside our LC101_Homework directory from the above example. We named that file `homework.js`. The absolute path for `homework.js` is `/Users/LaunchCodeStudent/Desktop/LC101_Homework` for Mac users and `C:\windows\Desktop\LC101_Homework` for Windows users. If the current directory is Desktop, then the relative path for `homework.js` is `/LC101_Homework` for Mac users and `\LC101_Homework` for Windows users.

If `homework.js` were in a different directory called `CoderGirl_Homework`, which is inside the Desktop directory, and the current directory was `LC101_Homework`, then we would use the `..` syntax in our relative path.

The relative path would then be `/../CoderGirl_Homework` for Mac users and `\..\CoderGirl_Homework` for Windows users.

Many programmers use paths to navigate through the filesystem in the terminal. We will discuss the commands to do so in the next section.

19.3 How to Do Stuff in the Terminal

19.3.1 Navigating the Terminal Window

Moving from a GUI to a CLI can be difficult when we are so used to dragging our files from one folder to another. One of the difficulties is simply figuring out where we are in the filesystem! Here are some key indicators that the terminal gives us to show where we are:

```
LaunchCode-Super-Computer:~ lcstaffmember$
```

This line is called the **prompt**. The prompt lets us know that the terminal is ready to accept commands. `LaunchCode-Super-Computer` is the name of the computer. The `~` tells us we are currently in the Home directory. The Home directory is the folder that contains everything in the computer. `lcstaffmember` is the username of the person who has logged onto the terminal. We will be typing all of our commands after the `$`.

As we navigate through our filesystem, the terminal will rarely output a line to let us know that the change has occurred. We have to keep our eye out on our prompt as we enter our commands. The name of the computer and the username will not change, however, the space where the `~` is, will. That indicates our current directory.

19.3.2 Basic Commands

There are many commands you can use in the terminal to move through the filesystem of your computer and projects.

Table 1: Basic Terminal Commands

Command	Result
<code>ls</code>	Lists all files and folders in the current directory.
<code>cd <new-directory></code>	Navigates from the current directory to <code>new-directory</code> .
<code>pwd</code>	Prints the path of the current directory.
<code>mkdir <new-folder></code>	Creates <code>new-folder</code> inside the current directory.
<code>touch <new-file></code>	Creates a file called <code>new-file</code> in the current directory.
<code>rm <old-file></code>	Removes <code>old-file</code> from the current directory.
<code>man <command></code>	Prints to the screen the manual pages for the command. This includes the proper syntax and a description of how that command works.
<code>clear</code>	Empties the terminal window of previous commands and output.
<code>cp <source-path> <target-path></code>	Copies the file or directory at <code>source-path</code> and puts it in the <code>target-path</code> .
<code>mv <source-path> <target-path></code>	Moves the file or directory at <code>source-path</code> from its current location to <code>target-path</code> .

Note

`rm` will permanently remove items from the computer and cannot be undone.

Beyond these basic commands, there are some shortcuts if you don't want to type out the full name of a directory or simply can't remember it.

Table 2: Directory Shortcuts

Shortcut	Where it goes
~	The Home directory
.	The current directory
..	The parent directory of the current directory

For an in-depth tutorial of how to use a CLI to move through your daily life, refer to the *terminal commands tutorial*.

19.3.3 Check Your Understanding

Question

What line in a CLI indicates that the terminal is ready?

- a. prompt
 - b. command
 - c. shell
 - d. There isn't a line that does that.
-

Question

Which shortcut takes you to the parent directory?

- a. .
 - b. ~
 - c. ..
-

19.4 Running Programs in the Terminal

Quickly navigating through our filesystems is just one benefit of using the terminal for programmers. We can also quickly run our code inside of the terminal to see the outputs.

The commands used to run a program in the terminal vary widely based on type of program you want to run. However, no matter what language you are coding in, the documentation will include, in some format, ways to run the program in the terminal.

Example

So far, in repl.it, we have been running our programs by hitting the “Run” button. If we type `node <file-name>` into our terminal, we would be doing the same thing as the “Run” button!

Let's say there is an error in our program like an infinite loop. How then do we get it to stop running so we can go back and fix our code?

In many cases, typing `ctrl+c` into the terminal will stop a process that is currently running. However, if that doesn't work, the `exit` command can also stop a currently running process.

19.5 Exercises: Terminal

1. If you haven't done so already, set up your command line environment with instructions from the *Setting Up Your Terminal* appendix.
2. Using your terminal, navigate to your Home directory using `cd ~`.
3. Use `ls` to view the contents of your Home directory.
4. Use `cd` to move into your Desktop directory. For most, the command to do this is `cd Desktop/` since the Desktop is most often a child of the Home directory.
5. In the terminal, use `mkdir` to create a folder on the Desktop called 'my_first_directory'. Look on your Desktop. Do you see it?
6. Use `cd my_first_directory/` to move inside that directory.
7. `pwd` to check your location.
8. There, make a file called 'my_first_file.txt' with `touch my_first_file`.
9. Open the file and write yourself a message!
10. Back in the terminal, list the contents of your current directory from the terminal with `ls`.
11. Make a copy of your 'my_first_file.txt' from its current spot to directly on the Desktop with `cp my_first_file.txt ../my_first_copy.txt`.
12. Move back out to your Desktop directory from the terminal with `cd ...`
13. Use `ls` in the terminal to verify your 'my_first_copy.txt' on your Desktop. Open it up. Is it the same as your first file?
14. Move your copied file into your 'my_first_directory' with `mv my_first_copy.txt my_first_directory/`.
15. Use `ls` to see that the copied file is no longer on your Desktop.
16. Type `cd my_first_directory/`, followed by `ls` to confirm that your copy has been moved into 'my_first_directory'.
17. `cd ..` to get back out to your Desktop.
18. Type `rm -r my_first_directory/` and do a visual check, as well as `ls` on your terminal, to verify that the directory has been removed.

WE BUILT THE INTERNET ON HTML

20.1 Background

When programmers make web pages, they want their pages to be beautiful, interactive, and fun. Programmers may use JavaScript to make their pages interactive, but JavaScript does not do much to define the structure and appearance of a web page. The next two chapters cover HTML and CSS, which are the two most common languages for structuring content and making it beautiful.

Before jumping in to learn HTML and CSS, we need to understand how web pages appear on screens. The process involves the browser and the server that hosts the code. You are probably very familiar with browsers as the tool that gives us access to the internet. However, programmers think of browsers a little differently. For them, the browser is what translates the code into a web page.

When you visit a web page in a browser, three main steps happen:

1. The browser sends a **request** to the server for the web page.
2. The server **responds** with the code that makes up the web page.
3. The browser takes the code and renders it to present the web page that the code creates.

When the browser renders the page, HTML outlines the structure of the page's content.

Note

In later chapters, request and response between browsers and servers will be covered in greater detail.

20.1.1 What is HTML?

Indicators of how HTML works are in its name. HTML is short for Hypertext Markup Language.

Hypertext is text that includes references to other text known as hyperlinks.

With coding languages, there is a family of languages called **markup languages**. Markup languages annotate the text of a document and define the structure. HTML is the markup language that defines the structure of hypertext.

HTML's two main components, elements and tags, are key to defining the structure of content.

20.1.2 HTML Elements

When a programmer creates a web page, they break the content down by type. They may outline a structure for the page on paper first, highlighting what each item is. With HTML, a programmer can add a lot of different types of content to a page. In this chapter, the focus is on headings, paragraphs, images, and more.

An **element** is a segment of an HTML page. Elements are oftentimes broken down by content type.

20.1.3 HTML tags

An HTML **tag** is the syntax that the computer processes to determine the type and content of an HTML element.

Tags surround the content within the element, so in all cases, programmers need to have opening and closing tags.

Each tag has the following structural elements:

1. < to start a tag and > to close it.
2. The type of element it is.
3. Optional additional specification about the element's appearance.
4. Closing tags include the same information as the opening tag with a / after the < bracket.

Here is an example of a line of HTML:

```
<element type>content</element type>
```

20.1.4 HTML Writing Style

Programmers write HTML different ways with different style guides and philosophies. **Semantic HTML** is not about the appearance of the web page, but about the specific meaning of the elements. Semantic HTML helps programmers communicate through code and may be easier to pick up at first. Programmers can make a paragraph larger than a heading. But by looking at the HTML, another programmer can understand which is the paragraph and which is the heading. Another benefit to semantic HTML is that it is easier for beginning programmers to visualize the end results. Some examples of semantic HTML tags are: <p>, <h1>, <h2> , and <div>.

Reminder

Making code work is important and so is making it easier for other programmers to read. Not every piece of code a programmer reads is something they wrote.

20.1.5 Check Your Understanding

Question

What does HTML stand for?

1. Happy Tickles Make Laughter
 2. Hypertext Markup Language
 3. Hypertext Mockup Language
 4. Hyperlink Markup Layout
-

20.2 HTML Structure

Programmers should follow certain rules about how to structure an HTML file. The rules about how to structure an HTML file and the tags used to lay out this structure are vital to the browser being able to render the page.

20.2.1 Structure Rules

When it comes to laying out the overarching structure of an HTML file, a programmer should follow 5 rules:

1. Every HTML file needs a DOCTYPE tag, specifying the HTML version used. When using the current version of HTML, the DOCTYPE tag is simple to remember as it is: `<!DOCTYPE html>`. This is one of few tags that does not require a closing tag.
2. The `<html>` tag denotes the beginning and end of the HTML the programmer has written.
3. The `<head>` tag contains data about the web page.
4. The `<body>` tag contains everything that appears on the page of the document.
5. The `<title>` tag goes in the `<head>` of the document and browsers require it. It gives the title of the webpage that appears in the tab.

Here is an example of the structure of an HTML page based off of these rules:

```
1 <!DOCTYPE html>
2   html
3     head
4       title My Web Page title
5       content
6     head
7   body
8     content
9   body
10  html
```

20.2.2 Document Head

So other than the title, what goes in the head of an HTML file? The head includes links to other files and other data about the document. Browsers do not display the content in the head.

Note

The head can also include some styling to make the page beautiful. How to do that is covered in the next chapter on CSS.

20.2.3 Document Body

After the programmer has written the head of the document, it is time to move on to the body of the document. The body of the document contains the content that appears on the web page. Within the `body` tags, programmers add images, text, and even code samples with different HTML tags. Content outside of the body will not appear on the page.

To make HTML more readable to other programmers, programmers write comments in HTML. When adding a comment, the programmer uses `<!--` to indicate the start and `-->` to end the comment, like so:

```
1 body
2 <!-- This is an important comment -->
3 body
```

Note

Spacing and tabs helps many programmers read through theirs and their colleagues' code. Be aware that doing so in HTML can effect how the browser renders the page in rare instances.

20.2.4 Check Your Understanding

Question

Which HTML tag does not require a closing tag?

1. title
 2. body
 3. head
 4. DOCTYPE
-

20.3 HTML Tags

Time to dive into learning about all the different tags for creating content! This page contains a helpful table of tags to know for beginning programmers to bookmark. This is by no means an exhaustive list of all HTML tags, but it is a good place to start.

20.3.1 Tags to Know

Tag Name	Code	Definition
Bold		When surrounding text, makes that text bold.
Emphasis		When surrounding text, makes that text italic.
Hyperlink	<a>	Creates hyperlinks.
Image		Denotes images.
Break	 	A single line break.
Paragraph	<p>	Creates a paragraph in text.
Section		Makes a section in text.
Division	<div>	Defines an area of the page.
Form	<form>	Creates a form for user input.
Unordered List		Creates an unordered list.
Ordered List		Creates an ordered list.
List element		Denotes an element of the list. This tag is used for both ordered and unordered lists.
Table	<table>	Creates a table on the page.
Heading Level One	<h1>	Creates a heading in the text.

Note

There are multiple headings in HTML going from h1 to h6. The headings get progressively smaller. A good rule of thumb is to have only one h1 in a web page and do not skip a level. Headings can be resized so there is no need to do so.

20.3.2 Tag Example

Here is an example of a basic web page utilizing some of the tags above with the HTML used to make the site.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Plant-Loving Astronauts  title
5     head
6   body
7     h1 Space Plants Are Cool  h1
8       p NASA discovers that plants can live in  b outer space  b . More innovations_
9         ←from this discovery to follow.  p
10        <!-- add images from NASA of these space plants -->
11      body
12    html
```

Space Plants Are Cool

NASA discovers that plants can live in **outer space**. More innovations from this discovery to follow.

20.3.3 Attributes

Programmers can add extra information beyond element type to HTML tags. Programmers add **attributes** to HTML tags for further specification about the element's appearance. Examples of attributes include the alignment of the element or alternate text to an image.

Programmers add attributes before the closing bracket in the opening tag, like so:

```
element attribute  "value" content  element
```

20.3.4 Attributes Example

Here is an example of a basic web page utilizing some of the tags above and appropriate attributes with the HTML used to make the site.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Plant-Loving Astronauts  title
5     head
6   body
7     h1 Space Plants Are Cool  h1
8       p NASA discovers that plants can live in  b outer space  b . More innovations_
9      ←from this discovery to follow.  p
10      img src  "space-flower.jpg" alt  "Flower floating in space."
11      <!-- This image was taken by NASA and is in the Public Domain -->
12    body
13  html
```

Space Plants Are Cool

NASA discovers that plants can live in **outer space**. More innovations from this discovery to follow.



The `` tag has two attributes that you will see a lot. `src` gives the location of the image that is being used and `alt` gives alternate text for screen reader users. For that reason, `alt` should be a concise description of what is going on in the image.

20.3.5 Check Your Understanding

Question

Which tag is used to make text italicized?

1. `b`
2. `i`
3. `em`
4. `br`

20.4 Exercises: HTML

Complete the HTML file for this simple webpage. Add lines to `index.html` that do the following.

1. Add a `h1` to the page that says “Why I Love Web Development”
2. Add an ordered list to the page with 3 reasons why you love web development.
3. Add a link to this page below your list.
4. Add a paragraph about the website you want to make with your web development superpowers!

This code block gives you a rough outline for how it might look.

```
1 <!DOCTYPE html>
2   html
3     head
4       head
5       body
6         <!-- h1 goes here -->
7         <!-- ol goes here -->
8         <!-- a goes here -->
9         <!-- p goes here -->
10        body
11      html
```

[repl.it](https://repl.it/@louisewilson/20.4-Exercises-HTML)

Note

repl.it has other HTML inside of the `index.html` file you will be editing. You should not be deleting any code only, only adding code to the file!

20.5 Studio: Making Headlines

20.5.1 Getting Ready: Developer Tools

As you’ve learned, debugging is an essential part of coding. When it comes to debugging web pages, browser developer tools are indispensable.

This studio requires you to use Firefox’s developer tools. In particular, you should be able to:

- Open Firebox’s dev tools
- Inspect an HTML element
- Modify an element’s HTML
- Explain the difference between the content displayed when using *View Source* and what you see in the *Inspector* tab

Note

Introduction to Professional Web Development in JavaScript

The full documentation for Firefox's developer tools covers these items, and much more.

20.5.2 Studio

Pick a news site ([The New York Times](#), for example), and use your browser's developer tools to modify one of the main articles to use a picture and text of your choosing.

Have fun with this, but be respectful of others and avoid overtly critical political or social commentary.

You might do something like this:



Fig. 1: A Sample Fake Article

Image URLs

When linking to an image, pay attention to the protocol of both the site you are modifying and of the image you are including. The protocol will be either `http` or `https`.

If the site loads over `https` and your image uses `http` then the image may not load properly due to browser security restrictions. You should try to add `s` to the image protocol, and if that doesn't work, look for another image.

If you want to use an image of your own that is not already available via the internet, here's how:

- Upload the photo to a [Dropbox](#) account
- View the photo on Dropbox and select *Share*, then *Get link*, then *Go to link*
- You should now be viewing the image on the Dropbox site. If the URL contains `?dl=0`, remove it. Add `?raw=1` to the end of the URL in the location bar of your browser and hit *Enter*. The URL should look something like this:

```
https://www.dropbox.com/sc/qc3htnhv7fb3i2x/AAC5OzECOyBynstMDWawCZhxa?raw=1
```

- Copy the URL of the loaded image. You can use this URL within any HTML.

20.5.3 Resources

- Using Firefox's Page Inspector
- Firefox DevTools Documentation

CHAPTER
TWENTYONE

STYLING THE WEB WITH CSS

21.1 What is CSS?

21.1.1 Background

As discussed in the previous chapter, HTML lays out the structure of a document. With HTML attributes, programmers can add some specification to tags. Yet, when programmers make pages with only HTML, the web pages look rather bland. When making a web site, the structure of the page elements is important, as is how those elements appear.

While HTML creates the structure and content of the page, CSS adds the styling to make it beautiful! Cascading Styling Sheets (**CSS**) is a style sheet language that allows programmers to add styling to web documents. With CSS, programmers can change background and font colors, the size of different elements, and many more things.

CSS works by applying style rules to different elements. A style rule could be: “Make this lettering purple” or “Make this font Helvetica”. CSS is a *cascading* style sheet language because the style rules apply based on a specific precedence, so the rules “cascade”.

Note

This book covers style rules and the order of precedence in greater detail in the third section of this chapter.

21.1.2 Check Your Understanding

Question

What kind of language is CSS? Check ALL that apply.

1. Markup Language
 2. Programming Language
 3. Style Sheet Language
 4. Coding Language
-

21.2 CSS Structure

21.2.1 Writing CSS

Programmers can change a lot of different styling using CSS **rules**. A rule includes the selector and a declaration block. A **selector** determines which elements will be affected by the rule. Inside the declaration block, programmers set CSS properties to specific values. CSS has a lot of different properties and it would be impossible to memorize them all.

```
selector {  
    declaration block  
}
```

CSS Selectors

CSS has three different selectors that the programmer can use to make their style choices.

The first one that most beginners start with is the **element selector**. Element refers to the HTML elements, so if the selector used is `p`, then the styling will apply to all paragraph elements.

The **id selector** is a specific id given to one element for CSS styling, for example when one paragraph on the web page needs to be bright pink.

The final selector is the **class selector**. A class is a group of HTML elements that need the same styling. The class name is determined by the programmer. The class name should be unique and have meaning like variable names.

Declaration Blocks

The declaration block is a series of initializations of style rules in CSS for a selector. Programmers can write CSS two different ways depending on where the CSS is in relation to the HTML document. We will go more in depth about the differences between CSS locations in the next section.

Here is an example of how to write the declaration block for internal and external CSS:

```
1 selector  
2  
3  
4  
5
```

For inline CSS, the declaration block is inside one line of HTML like so:

```
tag style "property:value;property:value;property:value;" content tag
```

Every property in CSS has a default value. For example, `font-color` defaults to “black”. For that reason, programmers only need to declare the CSS properties they want to change from the default.

Note

HTML elements also have a default appearance. When creating web pages, we should be aware of which elements are inline elements and which elements are block elements. Inline elements will not start a new line (such as ``, ``, and ``) and block display elements do (such as `<h1>`, `<div>`, and `<p>`).

CSS Examples

Here are three different examples of how we can use selectors to make the text in a paragraph pink.

Element Selector

Using the element selector to change the color of all `<p>` elements,

```
1 P  
2     pink  
3
```

Using the element selector will make all paragraph elements on the page have pink text.

Class Selector

Now, if a few of the paragraphs on the page are given the class `pink-paragraph` on the HTML document, like so: `<p class="pink-paragraph">content</p>`. To use the class selector in CSS, we would write something like:

```
1  
2     pink  
3
```

In CSS, the class selector is preceded by `.` ..

Id Selector

If one paragraph is going to have pink text, the id selector on the HTML document would look like: `<p id="pinkParagraph">content</p>`. In CSS, we would use the id selector to make the paragraph pink:

```
1  
2     pink  
3
```

In CSS, the id selector is preceded by `#`.

21.2.2 Linking CSS to HTML

To get started with CSS, programmers need to add CSS to HTML.

There are three different places to add CSS in an HTML file as indicated above:

1. External: The CSS is in a separate file linked to the HTML document in the `<head>`. External linking of CSS is great for when programmers have large quantities of CSS that apply to the whole page.

```
1 head  
2   title My Web Page title  
3   link rel "stylesheet" type "text/css" href "styles.css"  
4 head
```

`link` is an HTML tag that tells the browser to connect what is inside the linked file to the web page content. `rel`, `type`, `href` are all HTML attributes that are required to properly link CSS and let the browser know that CSS is what is in the file and where the file is. `rel` should be set to “stylesheet”, because it designates how the link relates to the page. `type` will be set to “text/css” for all stylesheets. `href` is where the programmer enters the path to the stylesheet that should be used for the page.

2. Document or internal: All CSS styling is inside the HTML file, but within the `<head>`. Internal use of CSS is great for when the programmer has a small amount of CSS that applies to the whole document.

```
<head>
    <title>My Web Page</title>
    <style>
        selector {
            declaration block
        }
    </style>
</head>
```

3. Inline: Programmers add CSS styling to individual tags. This is a good place to add some specific styling. There is no selector in inline CSS; instead, the `style` attribute is used. This is because the styling only applies to that one instance of the HTML tag.

```
tag style "declaration block" content tag
```

Order of Precedence

Because there is an order of precedence to the location of CSS, it is important to be able to add or change CSS in all three locations. Programmers use this to their advantage if they want to be very specific with overwriting some CSS for one element. Inline CSS is highest in precedence with internal CSS being next and then external CSS is lowest.

21.2.3 Check Your Understanding

Question

What is the order of precedence in CSS?

1. Internal > External > Inline
 2. Inline > Internal > External
 3. Inline > External > Internal
 4. External > Internal > Inline
-

21.3 CSS Rules

Below are some examples of common CSS properties and what they do. It is by no means an exhaustive list of CSS properties, but it is a good place to start.

21.3.1 Good CSS Properties to Know

CSS Property	Definition	Default Value
<code>font-size</code>	Changes the size of the font.	medium or 20px
<code>color</code>	Changes the text color.	black
<code>font-family</code>	Changes the font types.	Depends on the browser
<code>background-color</code>	Sets the color of the background of an element.	transparent
<code>text-align</code>	Aligns the text within an element.	left

21.3.2 CSS Example

Adding CSS to the HTML page about Space Plants is the logical next step in building a website about this cool discovery. The astronauts building the site used the `body`, `h1`, and `p` selectors to change some of the styling of those elements.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Plant-Loving Astronauts  title
5       style
6         body
7           background-color  cornflowerblue
8
9         h1
10        color  green
11
12        p
13          font-size  18px
14
15        style
16      head
17      body
18        h1 Space Plants Are Cool  h1
19        p NASA discovers that plants can live in  b outer space  b . More innovations_
→from this discovery to follow.  p
20        img src  "space-flower.jpg" alt  "Flower floating in space."
21        <!-- This image was taken by NASA and is in the Public Domain -->
22      body
23    html
```

Space Plants Are Cool

NASA discovers that plants can live in **outer space**. More innovations from this discovery to follow.



21.3.3 Check Your Understanding

Question

Find a CSS property and give its name, definition, and default value. Please do NOT use one of the ones above.

CHAPTER
TWENTYTWO

GIT MORE COLLABORATION

22.1 What is Git?

22.1.1 Version Control Systems

A version control system (VCS) is a system for tracking changes to a code base and storing each version. Version control systems assist programmers with keeping backups and a history of the revisions made to the code base over time. With that history, programmers can roll back to a version without a particular bug. A VCS also enables collaboration between programmers as they can work on different versions of a code base and share their work.

Git is one VCS and is prevalent amongst programmers and corporations.

A VCS has a **repository** or storage container for the code base. Repositories include the files within the code base, the versions over time and a log of the changes made. When a programmer updates the repository, it means they are making a **commit**.

22.1.2 Check Your Understanding

Question

What is a benefit of using a VCS?

22.2 Respositories and Commits

22.2.1 Create a Repository

To get started with a git repository, the programmer must first create one. To create a git repository, the programmer navigates to their project directory and uses the command `git init`, like so:

```
Students-Computer:~ student$ mkdir homework
Students-Computer:~ student$ cd homework
Students-Computer:homework student$ git init
      Initialized empty Git repository in /Users/student/homework/.git/
```

Now the programmer is ready to code away!

22.2.2 Making Commits

After a while, the programmer has made a lot of changes and saved their code files many times over. So when do they make a commit to their repository?

The general rule of thumb is that any time a significant change in functionality is made, a commit should be made.

If the programmer has created the Git repository and is ready to commit, they can do so by following the commit process.

Note

Git does have a simple commit command, however, making a proper commit requires that the programmers follow a longer procedure than just one command.

The procedure for making a commit to a Git repository includes 4 stages.

1. `git status` gives the programmer information about files that have been changed.
2. `git add` allows the programmers to add specific or all changed files to a commit.
3. `git commit` creates the new commit with the files that the programmer added.
4. `git log` displays a log of every commit in the repository.

If the steps above are followed correctly, the programmer will find their latest commit at the top of the log.

Here is how the process will look in the terminal:

```
Students-Computer:homework student$ git status
On branch master

Initial commit

Untracked files:
  use "git add <file>..." to include in what will be committed

    learning-git.js

nothing added to commit but untracked files present  use "git add" to track
Students-Computer:homework student$ git add .
Students-Computer:homework student$ git commit -m "My first commit"
  master root-commit 2c1e0af My first commit
  1 file changed, 1 insertion +
  create mode 100644 learning-git.js
Students-Computer:homework student$ git log
commit 2c1e0af9467217d76c7e3c48bcf9389ceaa4714b
Author: Student <lcl01.student@email.com>
Date:   Wed Apr 24 14:44:59 2019 -0500

  My first commit
```

To break down what happens in a commit even further:

When using `git status`, the output shows two categories: modified tracked files and modified untracked files. Modified tracked means that Git the file exists in the repository already, but is different than the version in the repository. Modified untracked means that it is a new file that is not currently in the repository.

`git add` adds files to the commit, but it does not commit those files. By using `git add ..`, all the modified files were added to the commit. If a programmer only wants add one modified file, they can do so.

`git commit` actually commits the files that were added to the repository. By adding `-m "My first commit"`, a comment was added to the commit. This is helpful for looking through the log and seeing detailed comments of the changes made in each commit.

`git log` shows the author of the commit, the date made, the comment, and a 40-character hash. This hash or value that is a key for Git to refer to the version. Programmers rarely use these hashes unless they want to roll back to that version.

22.2.3 Check Your Understanding

Question

What git command is NOT a part of the commit process?

1. `git add`
 2. `git log`
 3. `git status`
 4. `git push`
-

22.3 Remote Repositories

22.3.1 Local, Remote, Github, Oh My!

So far, the book has covered how to setup a Git repository on the local machine. But, one of the benefits of using a VCS was storage of backups. What happens to the code base if something happens to the machine. That is where remote repositories come in. Instead of keeping a Git repository only on a local machine, the code base is in a **remote repository** and the programmers working on it keep copies on their local machine.

To get started with remote repositories, create an account on [Github](#). From there, programmers can create a remote repository, view commit history, and report issues with the code.

22.3.2 Collaborating with Colleagues

What if a programmer wants to start collaborating with their colleagues on a new project? They might need to start with the work that one of their colleagues has already done. In this particular case, the programmer has to import the work that is being stored in an online repository onto their local machine.

They can clone a remote repository by using the `git clone <url>` command. Github and other online Git systems give users the option to clone the repository through HTTPS or SSH depending on how their Github profile is set up. The `<url>` of the command is where the programmer adds the url to the repository that they are cloning.

Note

Throughout this book, HTTPS will be used for cloning repositories.

22.3.3 Contributing to a Remote Repository

Now that the programmer has a profile on Github and a local copy of a remote repository, they start coding!

Once they create a new feature, it is time to make a commit. When working with a remote, the commit process has five steps:

1. `git status`
2. `git add`
3. `git commit`
4. `git push origin master`
5. `git log`

The fourth step uses the new command `git push` where the commit is pushed to the remote from the local. `origin` indicates that the commit does indeed go to the remote and `master` is the name of the branch that the commit goes to.

22.3.4 Check Your Understanding

Question

What is the new command for making a commit to a remote repository?

22.4 Branches

22.4.1 Branching in Git

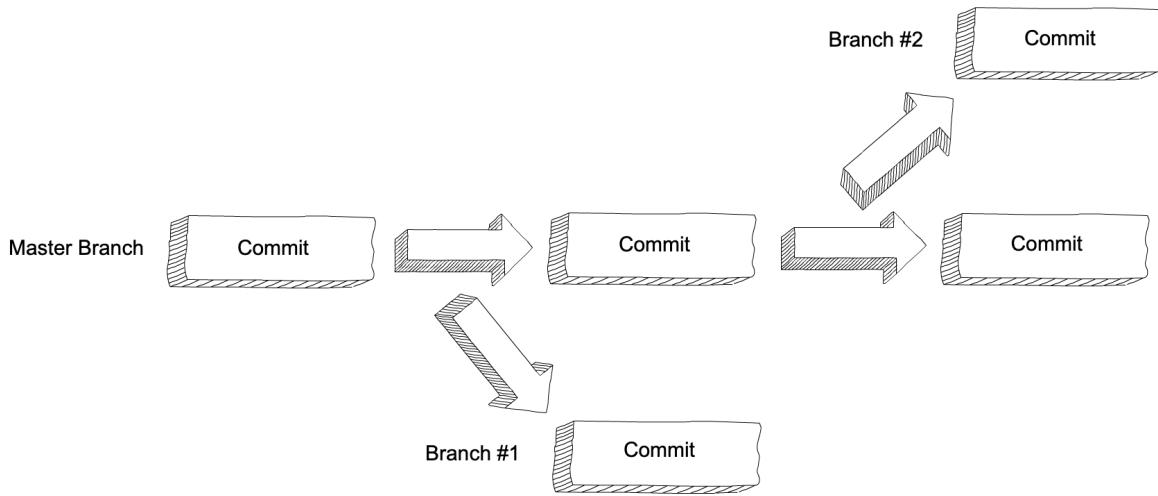
So far this book has talked about Git's ability to store different versions of a code base. What if two programmers want to work on different features of the code base at the same time? They may want to start with the same version and then one programmer wants to change the HTML and the other the CSS. It would not be effective for the two programmers to commit their changes to the repository at the same time. Instead, Git has branches. A **branch** is a separate version of the same code base. Like a branch on a tree, a branch in Git shares the same trunk as other branches, but is an individual. With branches, the two programmers could work on separate versions of the same website without interfering with each other's work. Besides collaboration, programmers use branches for storing and testing new features of software called feature branches.

In the previous section, when checking the status, the top line was `On branch master`. The master branch is the default branch of the repository. Many programmers keep the live version of their code in the master branch. For that reason, major work should be done in a new branch, so it doesn't impact the live software.

22.4.2 Creating a New Branch

A programmer is on master and they want to start building a new feature in a new branch. Their first step would be to create a new branch for their work.

To create a branch, the command is `git checkout -b <branch name>`. By using this command, not only is a new branch created, but also the programmer switches to their new branch.



22.4.3 Switching to an Existing Branch

If the branch already exists, the programmer may want to switch to that branch. To do so, the command is `git checkout <branch name>`.

22.4.4 Check Your Understanding

Question

What is a reason for creating a branch in Git?

22.5 Merging in Git

22.5.1 How to Merge

A **merge** in Git is when the code in two branches are combined in the repository. The command to merge a branch called `test` into `master` is `git merge test`. Before running the merge command, the programmer should make sure they are on the branch they want to merge into!

22.5.2 Merge Conflicts

This process is often seamless. In the example in the previous section, a programmer created a branch to change the HTML and the other programmer did the same to change the CSS. Because the two programmers changed different files, the merge of the updated HTML and updated CSS won't create a conflict. A **merge conflict** is when a change was made to the same line of code on both branches. Git doesn't know which change to accept, so it is up to the programmers to resolve it. Merge conflicts are minor on small applications, but can cause issues with large enterprise applications. Even though the thought of ruining software can be scary, every programmer deals with a merge conflict during their career. The best way to deal with a merge conflict is to face it head on and rely on teammates for support!

Ways to Avoid Merge Conflicts

Even though merge conflicts are normal in Git, it is also normal for programmers to want to do everything they can to avoid one. Here are some tips on how to avoid a merge conflict:

1. Git has a dry-run option for many commands. When a programmer uses that option, Git outputs what WILL happen, but doesn't DO it. With merging in Git, the command to run a dry-run and make sure there aren't any conflicts is `git merge --no-commit --no-ff <branch>`. `--no-commit` and `--no-ff` tell Git to run the merge without committing the result to the repository.
2. Before merging in a branch, any uncommitted work that would cause a conflict needs to be dealt with. A programmer can opt to not commit that work and instead **stash** it. By using the `git stash` command, the uncommitted work is saved in the stash and the repository is returned to the state at the last commit.

22.5.3 Check Your Understanding

Question

If a programmer is on the branch `test` and wants to merge a branch called `feature` into `master`, what steps should they take?

22.6 Exercises: Git

22.6.1 Working in a Local Repository

We will use our new terminal powers to move through the Git exercises.

1. In whichever directory you are keeping your coursework, make a new directory called `Git-Exercises` using the `mkdir` command.
2. Inside the `Git-Exercises` directory, initialize a new Git repository using `git init`.
3. Add a file called `exercises.txt` using the `touch` command in the terminal.
4. Commit your local changes using the `git commit` procedures.
5. Add "Hello World!" to the file called `exercises.txt`.
6. Commit your local changes following the same steps to you used for step 4.
7. View and screenshot the result when you use `git log`. Make note of what you see!

22.6.2 Setting up a Github Account

For our remote repositories, we will be using [Github](#).

To create your account, follow these steps:

1. Navigate to Github's site using the link above.
2. Sign up for an account on the homepage either by filling out the form or clicking the "Sign Up" button.
3. Once you have an account, you are ready to store your remote work.

22.7 Studio: Communication Log

22.7.1 Getting Ready: Code Together

Coding together allows you to work as a team so you can build bigger projects faster.

In this studio, we will practice the common Git commands used when multiple people work on the same code base.

You and a partner will begin by coding in tag-team shifts. By the end of the task you should have a good idea about how to have two people work on the same code at the same time. You will learn how to:

1. Quickly add code in pull + push cycles (*Important! This is the fundamental process!*)
2. Add a collaborator to a GitHub Project
3. Share *repositories* on GitHub
4. Create a *branch* in Git
5. Create a *pull request* in GitHub
6. Resolve merge conflicts (which are not as scary as they sound)

This lesson reinforces:

1. Creating repositories
2. Cloning repositories
3. Working with Git concepts: Staging, Commits, and Status

22.7.2 Overview

The instructor will discuss why GitHub is worth learning. You already know how to use a local Git repository with one branch, giving you the ability to move your code forward and backward in time. Working with branches on GitHub extends this ability by allowing multiple people to build different features at the same time, then combine their work. Pull requests act as checkpoints when code flows from branch to branch.

Students *must* pair off for this exercise. If you have trouble finding a partner, ask your TA for help.

22.7.3 Studio

We are going to simulate a radio conversation between the shuttle pilot and mission control.

First, find a new friend to share the activity.

You and your partner will alternate tasks, so designate one of you as **Pilot** and the other as **Control**. Even when it is not your turn to complete a task, read and observe what your partner is doing to complete theirs. The steps here mimic how a real-world collaborative Git workflow can be used within a project.

Warning

As you go through these steps, you'll be working with branches. It's very likely you will make changes to the code only to realize that you did so in the wrong branch. When this happens (and it happens to all of us) you can use `Git stash` to cleanly move your changes to another branch. Read about how to do so in our *Git Stash* tutorial.

Step 1: Create a New Repository

Control: Navigate to your development folder. Follow these instructions to create a new project.

```
$ mkdir communication-log  
$ cd communication-log  
$ git init
```

In that directory, open a new file `index.html` in the editor of your choice. Paste in this code:

```
1 html  
2 body  
3   p Radio check. Pilot, please confirm. p  
4 body  
5 html
```

Let's check that our html looks okay by opening it in a browser. Do this by selecting *File > Open File* in your web browser, and navigating to the location of your new HTML file. The URL will look something like this: `file:///Users/cheryl/Development/communication-log/index.html`.

Once you've seen this file in the browser, let's stage and commit it.

```
$ git status  
On branch master  
  
Initial commit  
  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
    index.html  
  
nothing added to commit but untracked files present (use "git add" to track)
```

The file is not staged. Let's add everything in this directory.

```
$ git add .  
$ git status  
On branch master  
  
Initial commit  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  
    new file:   index.html
```

We see that the file is staged. Let's commmit.

```
$ git commit -m 'Started communication log.'  
[master (root-commit) e1c1719] Started communication log.  
1 file changed, 5 insertions(+)  
create mode 100644 index.html  
$ git log  
commit 679de772612099c77891d2a3fab12af8db08b651  
Author: Cheryl <chrisbay@gmail.com>  
Date:   Wed Apr 5 10:55:56 2017 -0500  
  
        Started communication log.
```

Great! We've got our project going locally, but we're going to need to make it accessible for **Pilot** also. Let's push this project up to GitHub.

Step 2: Share Your Repository On GitHub

Control: Go to your GitHub profile in a web browser. Click on the “+” button to add a new repository (‘repo’).

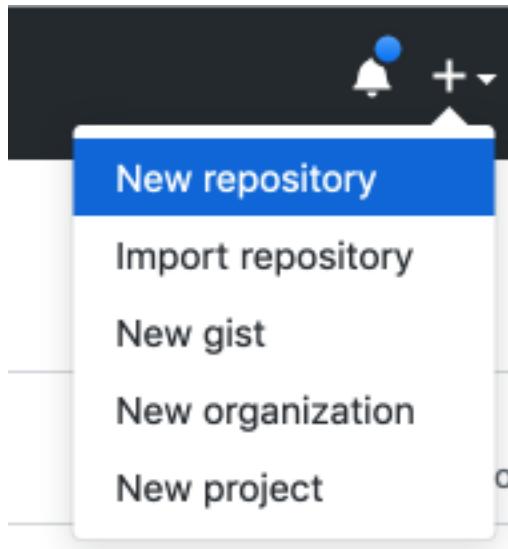


Fig. 1: The *New Repository* link is in the dropdown menu at top right on GitHub.

To create a new repository:

1. Fill in the name and description.
2. Uncheck *Initialize this repository with a README* and click *Create Repository*.

Note

If you initialize with a README, in the next step Git will refuse to merge this repo with the local repo. There are ways around that, but it's faster and easier to just create an empty repo here.

After clicking, you should see something similar to:

Now go back to your terminal and cut and paste the commands shown in the instructions on GitHub. These should be very similar to:

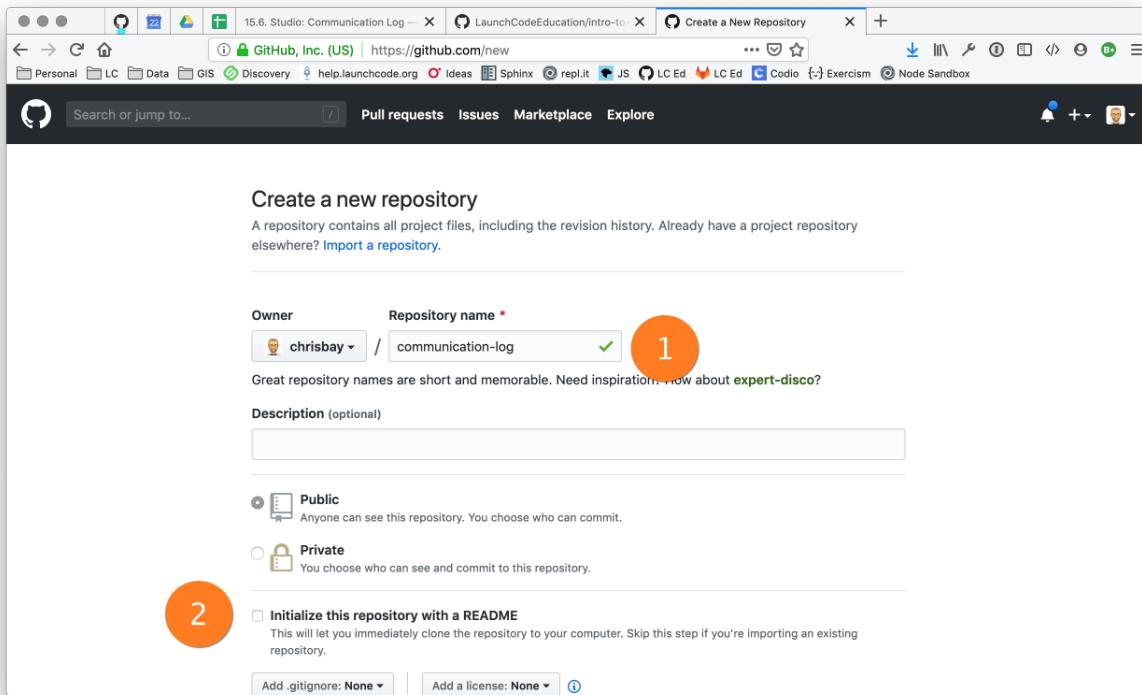


Fig. 2: Create a new repository in GitHub

```
$ git remote add origin https://github.com/chrisbay/communication-log.git  
$ git push origin master
```

Warning

Unless you've set up an SSH key with GitHub, make sure you've selected the HTTPS clone URL. If you're not sure whether you have an SSH key, you probably don't.

Now you should be able to confirm that GitHub has the same version as your local project. (File contents in browser match those in terminal). Click around and see what is there. You can read all your code through GitHub's web interface.

Step 3: Clone a Project from GitHub

Pilot: Go to Control's GitHub profile and find the communication-log repo. Click on the green *Clone or download* button. Use HTTPS (not SSH). Copy the url to your clipboard.

In your terminal, navigate to your development folder and clone down the repo. The command should look something like this.

```
$ git clone https://github.com/chrisbay/communication-log.git
```

Now you can respond to Control! Open the *index.html* file in your editor and add your response to mission control. Be creative—the communication can go anywhere! Just don't ask your partner what you should write. After you finish,

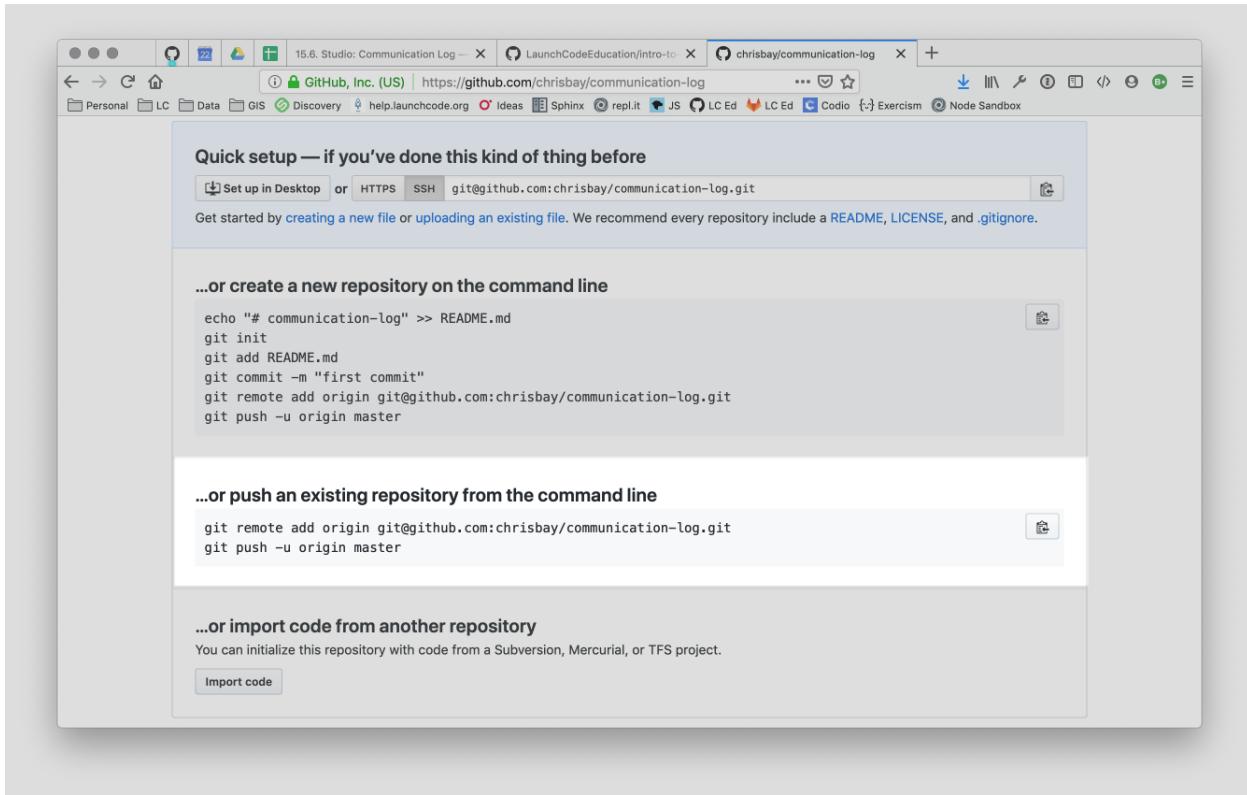


Fig. 3: Connecting to a repository in GitHub

commit your change.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
$ git add index.html
$ git commit -m 'Added second line to log.'
```

Now we need to push up your changes so Control can use them as well.

```
$ git push origin master
ERROR: Permission to chrisbay/communication-log.git denied to pilot.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Great error message! It let us know exactly what went wrong: Pilot does not have security permissions to write to Control's repo. Let's fix that.

Step 4: Add A Collaborator To A GitHub Project

Control: In your web browser, go to your *communication-log* repo. Click the *Settings* button then click on *Collaborators*. Enter in Pilot's GitHub username and click *Add Collaborator*.

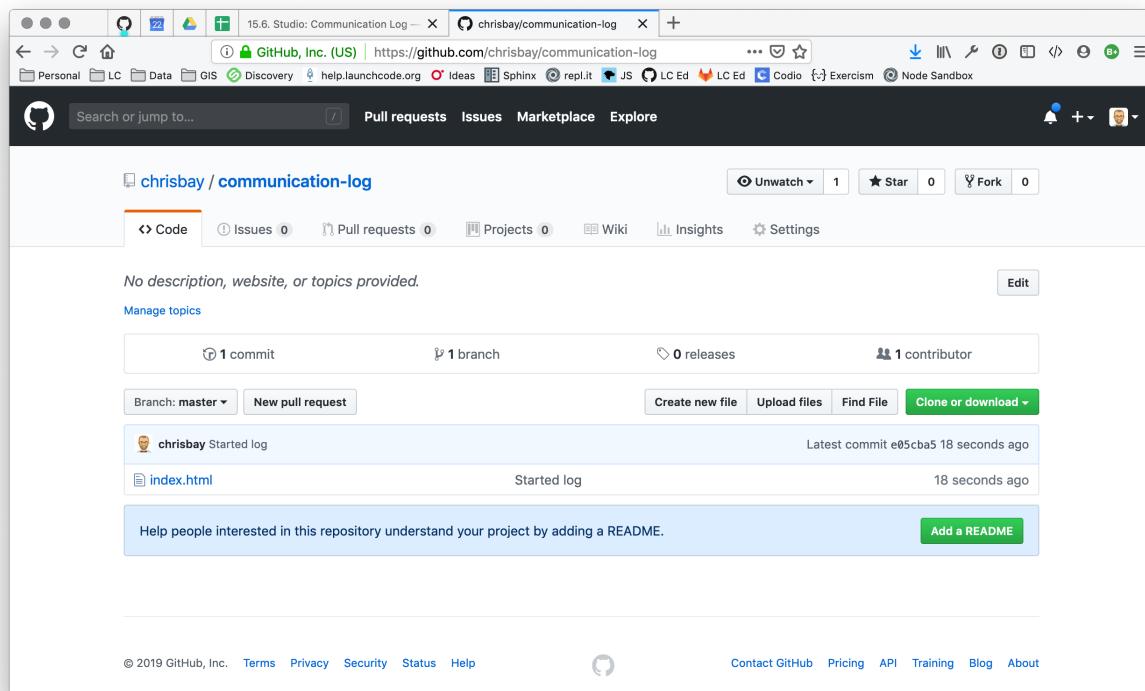


Fig. 4: A repository with one commit in GitHub

Step 5: Join the Project and Push

Piolt: You should receive an email invitation to join this repository. View and accept the invitation.

Note

If you don't see an email (it may take a few minutes to arrive in your inbox), check your Spam folder. If you still don't have an email, visit the repository page for the repo that Control created (ask them for the link), and you'll see a notification at the top of the page.

Now let's go enter that command again to push up our code.

```
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.01 KiB | 0 bytes/s, done.
Total 9 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), completed with 8 local objects.
To git@github.com:chrisbay/communication-log.git
  511239a..679de77  master -> master
```

Anyone reading the HTML through GitHub's browser interface should now see the new second line.

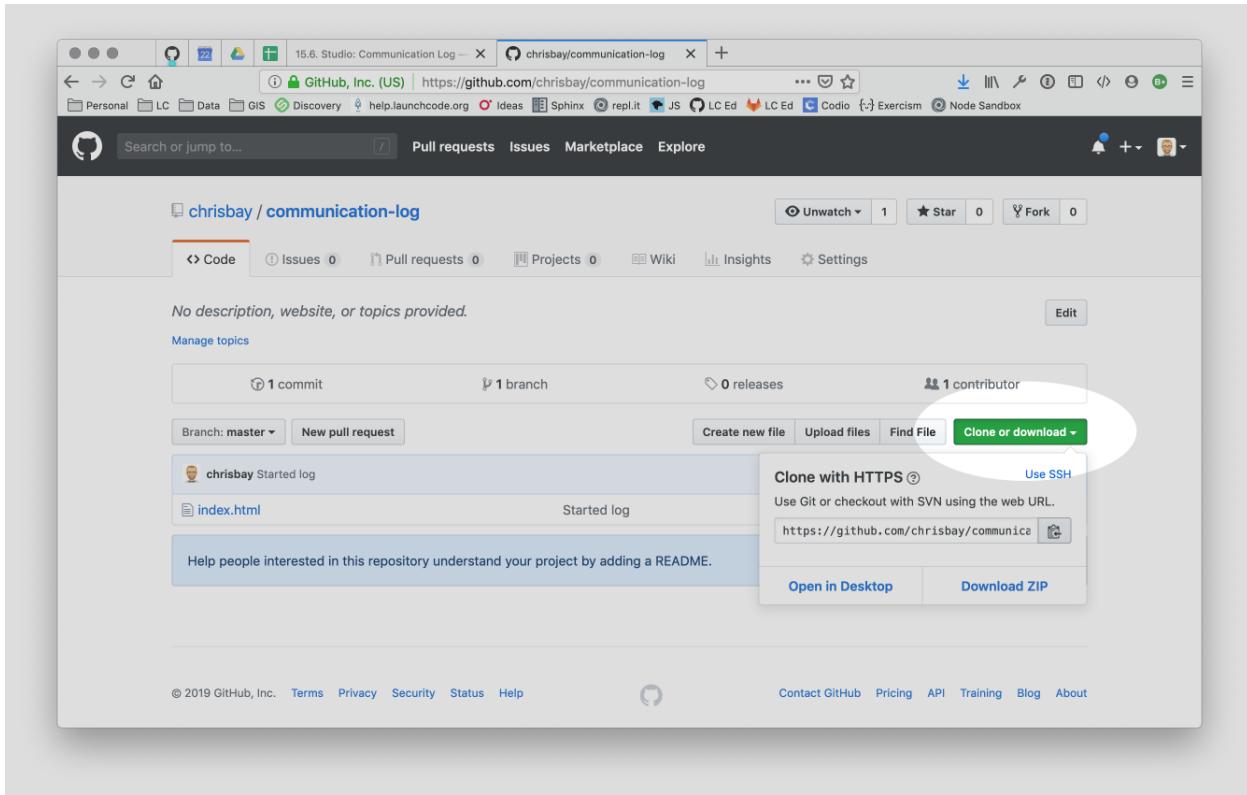


Fig. 5: Cloning a repository in GitHub

Step 6: Pull Pilot's Line and Add Another Line

Control: You might notice you don't have the second line of code in your copy of the project on your computer. Let's fix that. Go to the terminal and enter this command to pull down the updated code into your local git repository.

```
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:chrisbay/communication-log
  e0de62d..e851b7e  master      -> origin/master
Updating e0de62d..e851b7e
Fast-forward
 index.html | 1 +
 1 file changed, 1 insertion(+)
```

Now, in your editor, add a third line to the communication. Then add, commit, and push it up.

You can have your story go anywhere! Try to tie it in with what the pilot wrote, without discussing with them any plans on where the story will go.

Step 7: Do It Again: Pull, Change, and Push!

Pilot: You might notice now *you* don't have the third line on your computer. Go to the terminal and enter this command to pull in the changes that Control just made.

Introduction to Professional Web Development in JavaScript

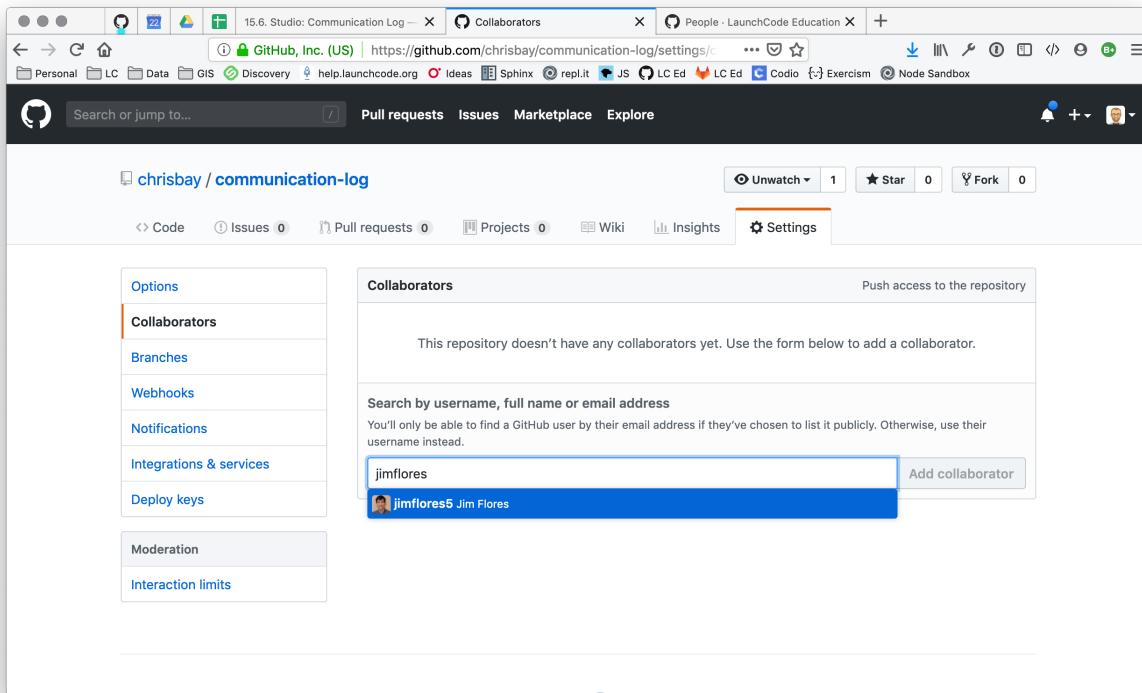


Fig. 6: Add a collaborator to your repo in GitHub

```
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:chrisbay/communication-log
  e851b7e..167684c  master      -> origin/master
Updating e851b7e..167684c
Fast-forward
 index.html | 1 +
 1 file changed, 1 insertion(+)
```

Now add a fourth line to the log. Again, be creative, but no planning!

Then add, commit, and push your change.

You can both play like this for a while! Feel free to repeat this cycle a few times to add to the story.

Step 8: Create a Branch In Git

This workflow is a common one in team development situations. You might wonder, however, if professional developers sit around waiting for their teammates to commit and push a change before embarking on additional work on their own. That would be a drag, and thankfully, there is a nice addition to this workflow that will allow for simultaneous work to be carried out in a reasonable way.

Pilot: While Control is working on an addition to the story, let's make another change simultaneously. In order to do that, we'll create a new branch. Recall that a branch is a separate "copy" of the codebase that you can commit to

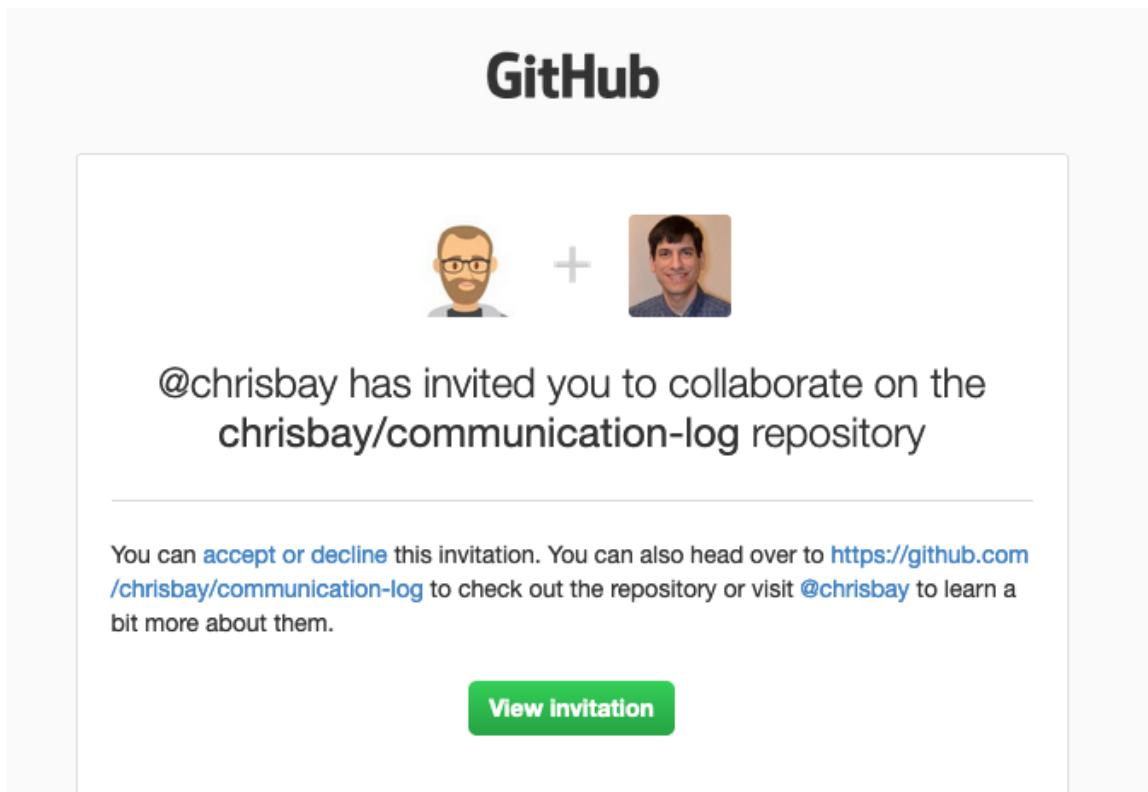


Fig. 7: Invited to collaborate email in GitHub

Introduction to Professional Web Development in JavaScript

without affecting code in the `master` branch.

```
$ git checkout -b open-mic  
Switched to a new branch 'open-mic'
```

This command creates a new branch named `open-mic`, and switches your local repository to use that branch.

Create a new file named `style.css` and add the following rules:

```
1 body  
2   color white  
3   background-color black  
4
```

Then link it in `index.html`. It should look something like this:

```
1 html  
2   head  
3     link rel "stylesheet" type "text/css" href "style.css"  
4   head  
5   body  
6     p Radio check. Pilot, please confirm. p  
7     ... your content here  
8   body  
9 html
```

Now stage and commit these changes.

```
$ git add .  
$ git commit -m 'Added style.css'  
$ git push origin open-mic
```

Note that the last command is a bit different than what we've used before (`git push origin master`). The final piece of this command is the name of the branch that we want to push to GitHub.

You and your partner should both now see a second branch present on the GitHub project page. To view branches on GitHub, select *Branches* from the navigation section just below the repository title.

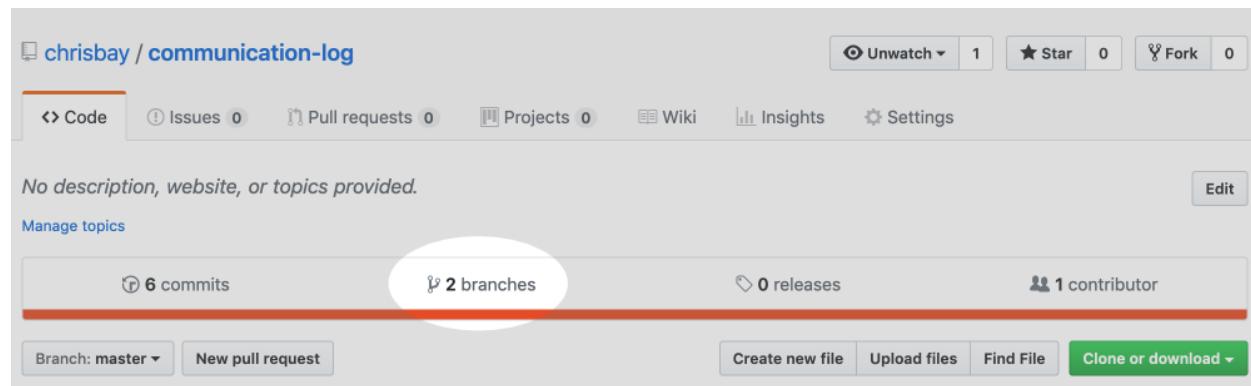


Fig. 8: Branches Button in GitHub

In your terminal, you can type this command to see a list of the available branches:

```
$ git branch  
* open-mic  
master
```

Note that creating and being able to see a branch in your local repository via this command does NOT mean that the branch is on GitHub. You'll need to push the branch for it to appear on GitHub.

Note

The * to the left of `open-mic` indicates that this is the active branch.

Great! Now let's show the other player your work in GitHub and ask them to merge it in to the main branch.

Create a Pull Request In GitHub

Pilot: If you haven't already, in your browser, go to the GitHub project and click on *Branches* and make sure you see the new branch name, *open-mic*.

The screenshot shows the GitHub 'Branches' page. At the top, there is a navigation bar with tabs: 'Overview' (which is selected and highlighted in blue), 'Yours', 'Active', 'Stale', and 'All branches'. To the right of the tabs is a search bar labeled 'Search branches...'. Below the navigation bar, there are three sections: 'Default branch' (showing the 'master' branch), 'Your branches' (showing the 'open-mic' branch), and 'Active branches' (also showing the 'open-mic' branch). Each branch entry includes the branch name, the last update time ('Updated 3 minutes ago by chrisbay' or 'Updated 2 minutes ago by Jim Flores'), and a 'Default' button next to the master branch. On the far right of each section, there is a 'New pull request' button. The 'Your branches' section has a small number '1|1' indicating one branch. The 'Active branches' section also has a small number '1|1'.

Fig. 9: Branches Page in GitHub

Click *New Pull Request* to begin the process of requesting that your changes in the `open-mic` branch be incorporated into the `master` branch. Add some text in the description box to let Control know what you did and why.

Note that the branch selected in the *base* dropdown is the one you want to merge *into*, while the selected branch in the *compare* dropdown is the one you want to merge *from*.

This is what an opened pull request looks like:

Step 10: Make a Change in the New Branch

Control: You will notice that you do not see the new `style.css` file locally. Type this command to see what branches are on your local computer:

```
$ git branch  
* master
```

If you want to work with the branch before merging it in, you can do so by typing these commands:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

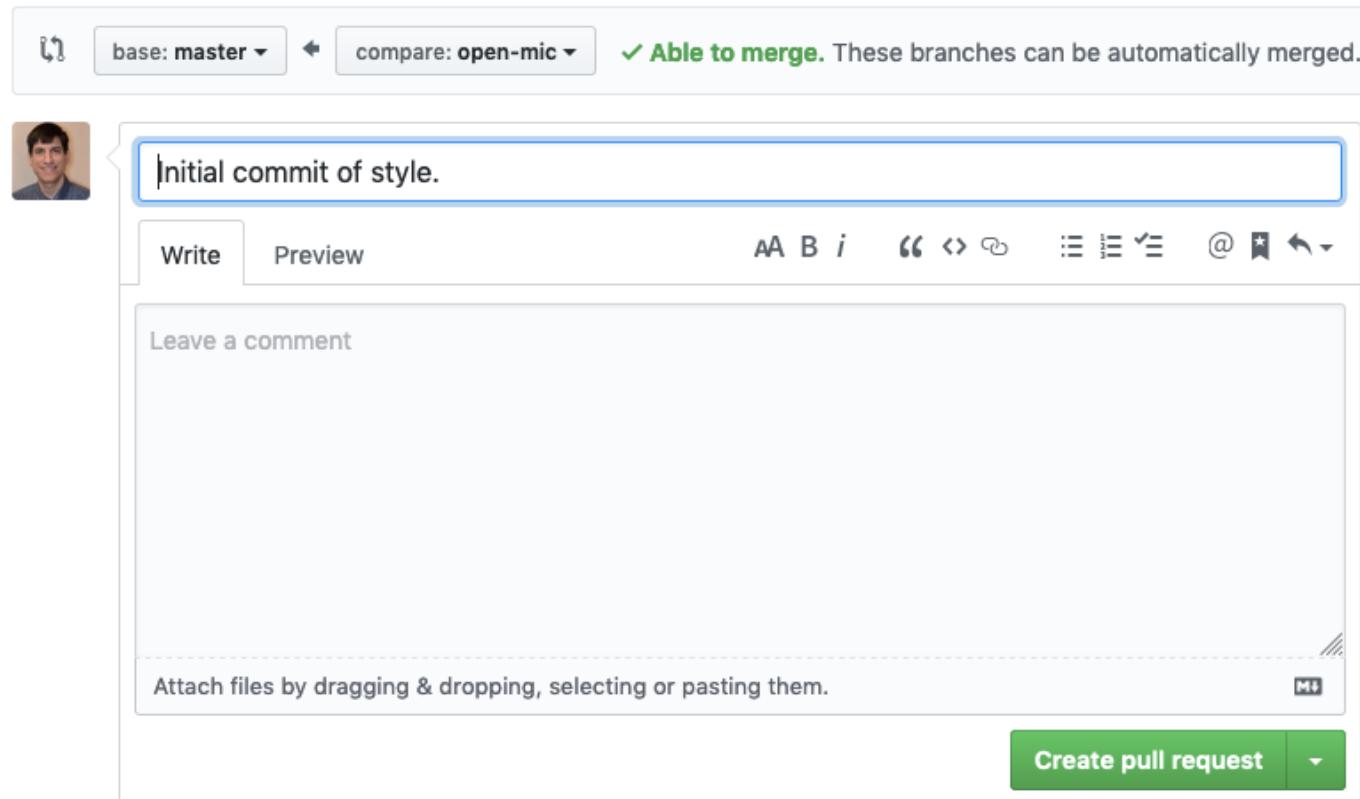


Fig. 10: Open a PR in GitHub

Initial commit of style. #1

 Open jimflores5 wants to merge 2 commits into `master` from `open-mic` 

 Conversation 0  Commits 2  Checks 0  Files changed 2

 jimflores5 commented 4 minutes ago Collaborator +  ...

No description provided.

 Jim Flores and others added some commits 9 minutes ago

-  Initial commit of style. 4db4f70
-  Updates font styles 8f4a278

Add more commits by pushing to the `open-mic` branch on [chrisbay/communication-log](#).

 **Continuous integration has not been set up**
Several apps are available to automatically catch bugs and enforce style.

 **This branch has no conflicts with the base branch**
Merging can be performed automatically.

Merge pull request ▾ You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Fig. 11: An open PR in GitHub

```
$ git fetch origin open-mic
...
$ git branch
open-mic
* master
```

```
$ git checkout open-mic
Switched to branch 'open-mic'
Your branch is up-to-date with 'origin/open-mic'.
```

Make a change, commit, and push this branch—you will see that the pull request in GitHub is updated to reflect the changes you added. The context in the description box is NOT updated, however, so be sure to add comments to the pull request to explain what you did and why.

Now switch back to the `master` branch:

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

You will see your files no longer have the changes made in the `open-mic` branch. Let's go merge those changes in, so that the ``master`` branch adopts all the changes in the `open-mic` branch.

Step 11: Merge the Pull Request

Control: Go to the repo in GitHub. Click on *Pull Requests*.

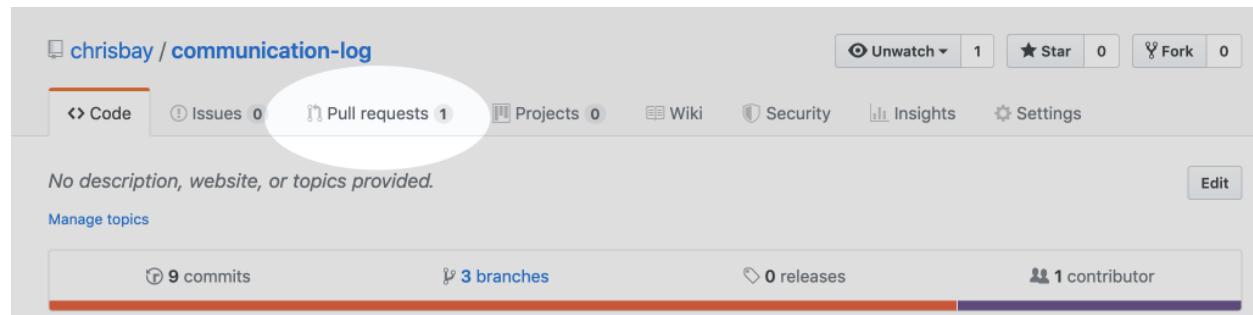


Fig. 12: PR Open in GitHub

Explore this page to see all the information GitHub shows you about the pull request.

When you're happy with the changes, merge them in. Click *Merge Pull Request* then *Confirm Merge*.

Upon a successful merge, you should see a screen similar to the following:

The changes from `open-mic` are now in the `master` branch, but only in the remote repository on GitHub. You will need to pull the updates to your `master` for them to be present locally.

```
$ git checkout master
$ git pull origin master
```

Git is able to merge these files on its own.

Initial commit of style. #1

 **jimflores5** wants to merge 2 commits into `master` from `open-mic` 

 Conversation 0  Commits 2  Checks 0  Files changed 2

 **jimflores5** commented 4 minutes ago

 +  ...

No description provided.

 **Jim Flores** and others added some commits 9 minutes ago

 **Initial commit of style.** 4db4f70

 **Updates font styles** 8f4a278

Add more commits by pushing to the `open-mic` branch on **chrisbay/communication-log**.



 **Continuous integration has not been set up**

Several apps are available to automatically catch bugs and enforce style.



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Fig. 13: Merge a Pull Request in GitHub

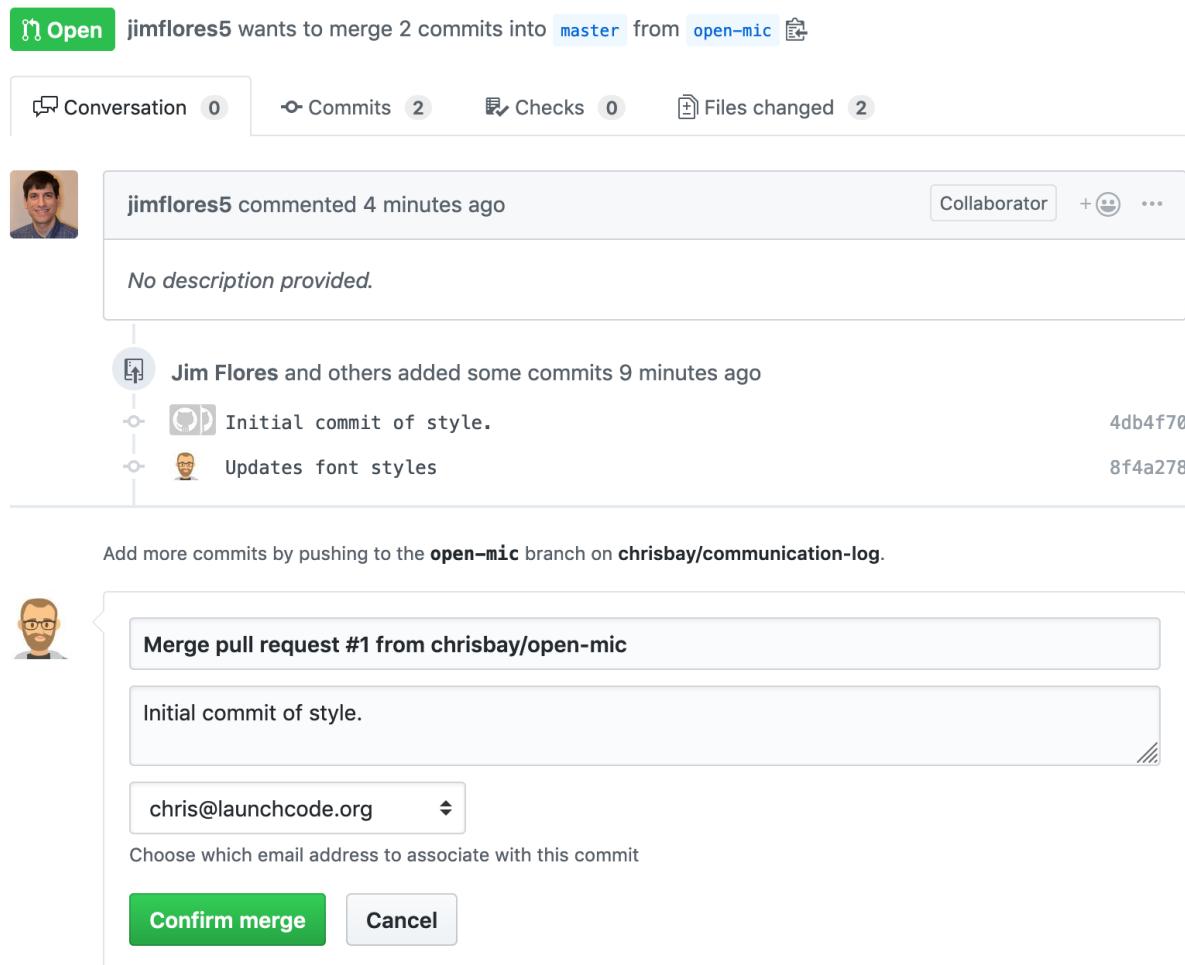


Fig. 14: Confirm PR Merge in GitHub

Initial commit of style. #1

Merged chrisbay merged 2 commits into master from open-mic just now

Conversation 0 Commits 2 Checks 0 Files changed 2

jimflores5 commented 4 minutes ago

No description provided.

Jim Flores and others added some commits 9 minutes ago

Initial commit of style. 4db4f70

Updates font styles 8f4a278

chrisbay merged commit a3f949c into master just now

Pull request successfully merged and closed

You're all set—the open-mic branch can be safely deleted.

Delete branch

Write Preview

Leave a comment

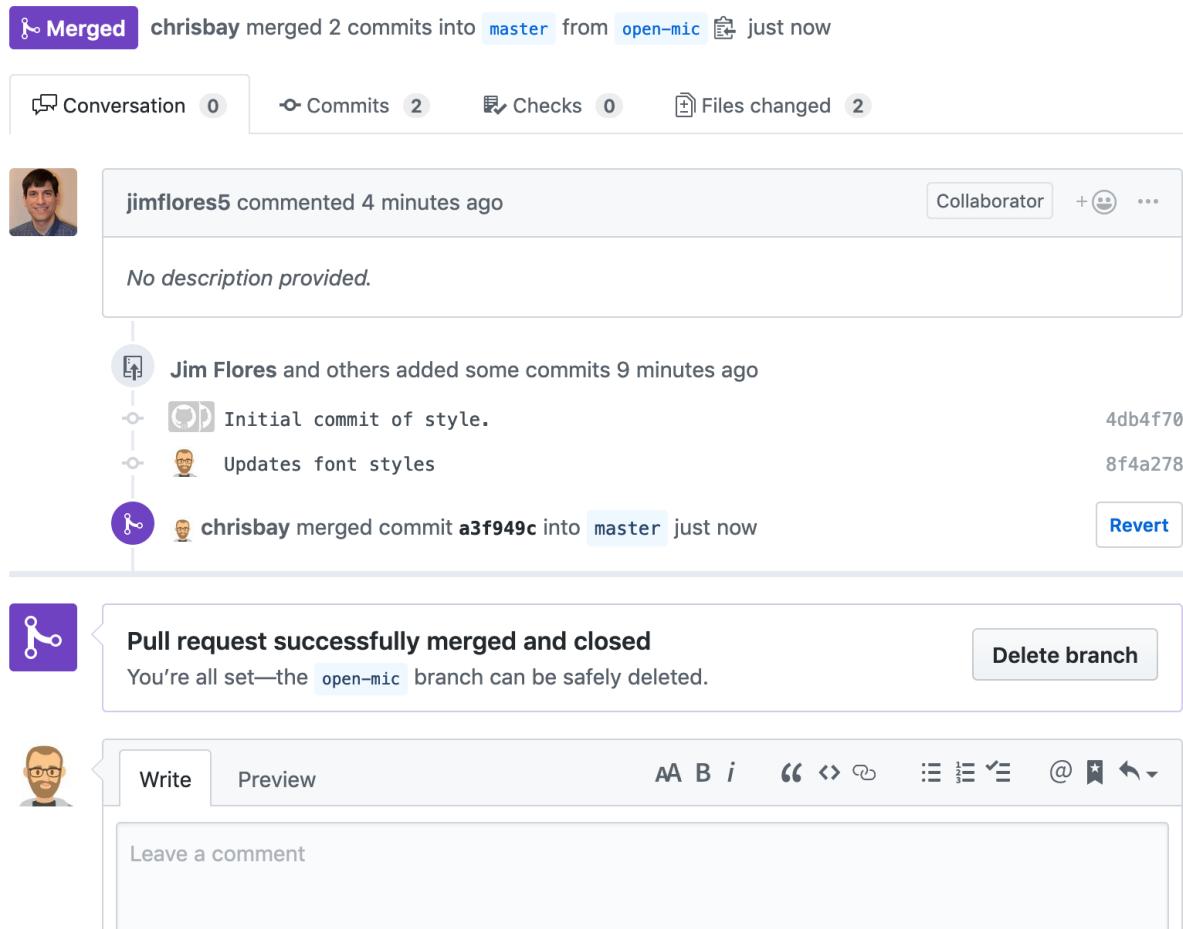


Fig. 15: PR Merged in GitHub

Step 12: Merge Conflicts!

When collaborating on a project, things won't always go smoothly. It's common for two people to make changes to the same line(s) of code, at roughly the same time, which will prevent Git from being able to merge the changes together.

Fig. 16: Git Merge Conflicts

This isn't such a big deal. In fact, it's very common. To see how we can handle such a situation, we'll intentionally create a merge conflict and then resolve it.

Pilot: Let's change something about the style file. Our HTML is looking pretty plain, so let's pick a nice font and add some margins.

First, switch back to the master branch.

```
$ git checkout master
```

Let's change our font. To do so, add this link to your `index.html` file, right after the first stylesheet link:

```
link href="https://fonts.googleapis.com/css?family=Satisfy" rel="stylesheet"
```

And spice up your `style.css` file to look like this:

```
1 body  
2   color white  
3   background-color #333  
4   font-size 150%  
5   font-family 'Satisfy' cursive  
6   margin 5em 25%  
7
```

The result:

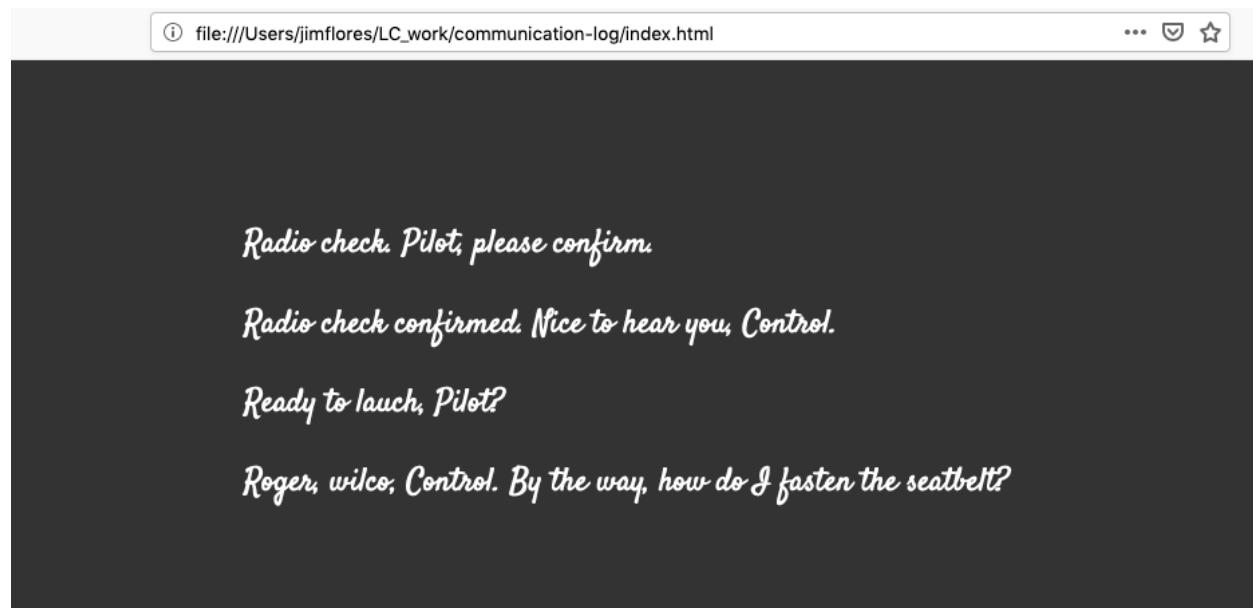


Fig. 17: Satisfying!

Stage and commit your changes and push them up to GitHub. If you don't remember how to do this, follow the instructions above. Make sure you're back in the `master` branch! If you're still in `open-mic`, then your changes will be isolated, and you won't get the merge conflict you need to learn about.

Meanwhile...

Control: Let's change something about the style file that Pilot just edited. Change it to look like this:

```
1 body
2   color white
3   background-color black
4   font-family 'Sacramento' cursive
5   font-size 32px
6   margin-top 5%
7   margin-left 20%
8   margin-right 20%
9
```

Don't forget to link the new font in your `index.html` file, after the other link:

```
link href "https://fonts.googleapis.com/css?family=Sacramento" rel "stylesheet"
```

Commit your changes to branch `master`.

Step 13: Resolving Merge Conflicts

Control: Try to push your changes up to GitHub. You should get an error message. How exciting!

```
$ git push origin master

To git@github.com:chrisbay/communication-log.git
! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'git@github.com:chrisbay/communication-log.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

There's a lot of jargon in that message, including some terminology we haven't encountered. However, the core of the message is indeed understandable to us: "Updates were rejected because the remote contains work that you do not have locally." In other words, somebody (Pilot, in this case), pushed changes to the same branch, and you don't have those changes on your computer. Git will not let you push to a branch in another repository unless you have incorporated all of the work present in that branch.

Let's pull these outstanding changes into our branch and resolve the errors.

```
$ git pull
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 4 (delta 1), pack-reused 0
Unpacking objects: 100% (4/4), done.
From github.com:chrisbay/communication-log
  7d7e42e..0c21659  master    -> origin/master
Auto-merging style.css
CONFLICT (content): Merge conflict in style.css
Auto-merging index.html
```

(continues on next page)

(continued from previous page)

```
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Since Pilot made changes to some of the same lines you did, Git was unable to automatically merge the changes.

The specific locations where Git could not automatically merge files are indicated by the lines that begin with CONFLICT. You will have to edit these files yourself to incorporate Pilot's changes. Let's start with style.css.

```
style.css — communication-log
index.html style.css

1 body [ ]
2 |   color: white;
3 <<<<< HEAD (Current Change)
4   background-color: black;
5   font-family: 'Sacramento', cursive;
6   font-size: 32px;
7   margin-top: 5%;
8   margin-left: 20%;
9   margin-right: 20%;
10 =====
11   background-color: #333;
12   font-size: 150%;
13   font-family: 'Satisfy', cursive;
14   margin: 5em 25%;
15 >>>>> a48e8a751c48fc1ad49150188a74add8ca923112 (Incoming Change)
16 [ ]
17
```

Fig. 18: Merge conflicts in style.css, viewed in VS Code

At the top and bottom, there is some code that could be merged without issue.

Between the <<<<< HEAD and ===== symbols is the version of the code that exists locally. These are *your* changes.

Between ===== and >>>>> a48e8a75... are the changes that Pilot made (the hash a48e8a75... will be unique to the commit, so you'll see something slightly different on your screen).

Let's unify our code. Change the CSS to look like this, making sure to remove the Git markers so that only valid CSS remains in the file.

```
1 body
2   color white
3   background-color black
4   font-family 'Sacramento' cursive
5   font-size 150%
6   margin 5em 25%
```

Tip

Like many other editors, VS Code provides fancy buttons to allow you to resolve individual merge conflicts with a single click. There's nothing magic about these buttons; they do the same thing that you can do by directly editing the

file.

You will need to do the same thing for the `index.html` file. You only need the link for the Sacramento font, not the Satisfy font. Then stage, commit, and push your changes; you should not see an error message this time.

Step 14: Pulling the Merged Code

Pilot: Meanwhile, Pilot is sitting at home, minding their own business. A random `git status` seems reassuring:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Your local Git thinks the status is quo. Little does it know that up at GitHub, the status is not quo. We'd find this out by doing either a `git fetch`, or if we just want the latest version of this branch, `git pull`:

```
$ git pull
remote: Counting objects: 13, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 13 (delta 4), reused 13 (delta 4), pack-reused 0
Unpacking objects: 100% (13/13), done.
From Github.com:chrisbay/communication-log
  0c21659..e0de62d  master      -> origin/master
Updating 0c21659..e0de62d
Fast-forward
 index.html | 3 +++
 style.css   | 4 +---
 2 files changed, 4 insertions(+), 3 deletions(-)
```

Great Scott! Looks like Control changed both `index.html` and `style.css`. Note that *Pilot* didn't have to deal with the hassle of resolving merge conflicts. Since Control intervened, Git assumes that the team is okay with the way they resolved it, and *fastforwards* our local repo to be in sync with the remote one. Let's look at `style.css` to make sure:

```
1 body
2   color white
3   background-color black
4   font-family 'Sacramento' cursive
5   font-size 150%
6   margin 5em 25%
7
```

Step 15: More Merge Conflicts!

Let's turn the tables on the steps we just carried out, so Pilot can practice resolving merge conflicts.

1. **Control and Pilot:** Confer to determine the particular lines in the code that you will both change. Make different changes in those places.
2. **Control:** Stage, commit, and push your changes.
3. **Pilot:** Try to pull in Control's changes, and notice that there are merge conflicts. Resolve these conflicts as we did above (ask Control for help, if you're uncertain about the process). Then stage, commit, and push your changes.

4. **Control:** Pull in the changes that Pilot pushed, including the resolved merge conflicts.

Merge conflicts are a part of the process of team development. Resolve them carefully in order to avoid bugs in your code.

Resources

- [Git Branching - Basic Branching and Merging](#)
- [Adding Another Person To Your Repository](#)
- [Resolving Conflicts In the Command Line](#)

CHAPTER
TWENTYTHREE

THE DOM AND EVENTS

23.1 JavaScript and the Browser

23.1.1 Taking JavaScript on the Web

So far, we have created web pages with HTML and CSS. These pages have been **static**, meaning that the page appears the same each time it loads. However, you may find that you want to create a web page that changes after it's been loaded. In order to create such a page, you would use JavaScript. Web pages that can change after loading in the browser are called **dynamic**. This is useful to programmers and users alike because they can interact with an application without refreshing the page. Having to constantly refresh the page would be a poor experience for the user and JavaScript helps programmers alleviate this burden.

Example

When you are on a social media page, you may like someone's post. When you do like their post, you may notice that several things happen. The counter of how many likes the post has received increases by one and the like button may change color to indicate to you that you liked the post. This is an example of how JavaScript could be used to create an application that dynamically updates without the page having to be refreshed.

We have been running all of our JavaScript code in Node.js, but now it is time to use JavaScript in the browser to make dynamic web pages. Node.js, or just Node, is a JavaScript interpreter with access to lots of different JavaScript libraries. Each browser has their own engine for running JavaScript. JavaScript run in the browser is called client-side JavaScript. Firefox uses an engine called Spider Monkey to run client-side JavaScript.

23.1.2 The `<script>` Tag

In the HTML chapter, we learned that an HTML page is made up of elements that are written as tags. Those elements have different purposes. The `script` element's purpose is to include JavaScript into the web page. A `<script>` tag can contain JavaScript code inside of it or reference an external JavaScript file.

JavaScript Console

Using the Developer Tools, you can access a JavaScript console. There, you can mess around with fun JavaScript statements or you can use it to see the outputs of the client-side JavaScript you have written.

Inline JavaScript

Example

Notice the `<script>` tag below contains JavaScript code that will be executed by the browser.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Embedded JavaScript Example  title
5       script
6         // JavaScript code goes here!
7           "Hello from inside the web page!"
8       script
9     head
10    body
11      contents
12    body
13  html
```

Console Output

```
Hello from inside the web page!
```

External JavaScript

Some programmers have large amounts of JavaScript to add to an HTML document. Using an external JavaScript file can help in these cases. You can still use the `<script>` tag to include the JavaScript file within the HTML document. In this case, you'll need to use the `src` attribute for the path to the JavaScript file.

Example

This is how the HTML file for the web page might look if we wanted to link an external JavaScript file.

```
1 <!DOCTYPE html>
2   html
3     head
4       title External JavaScript Example  title
5       script src  "myjs.js"  script
6     head
7     body
8       <!-- content -->
9     body
10    html
```

Then the JavaScript file, `myjs.js` might look something like this.

```
1 // JavaScript code goes here!
2   "Hello from inside the web page"
```

Note

You can use the `<script>` tag to reference JavaScript files hosted on external servers. Some of these JavaScript files will be files that you have not written yourself but you will want to include in your application.

23.1.3 Check Your Understanding

Question

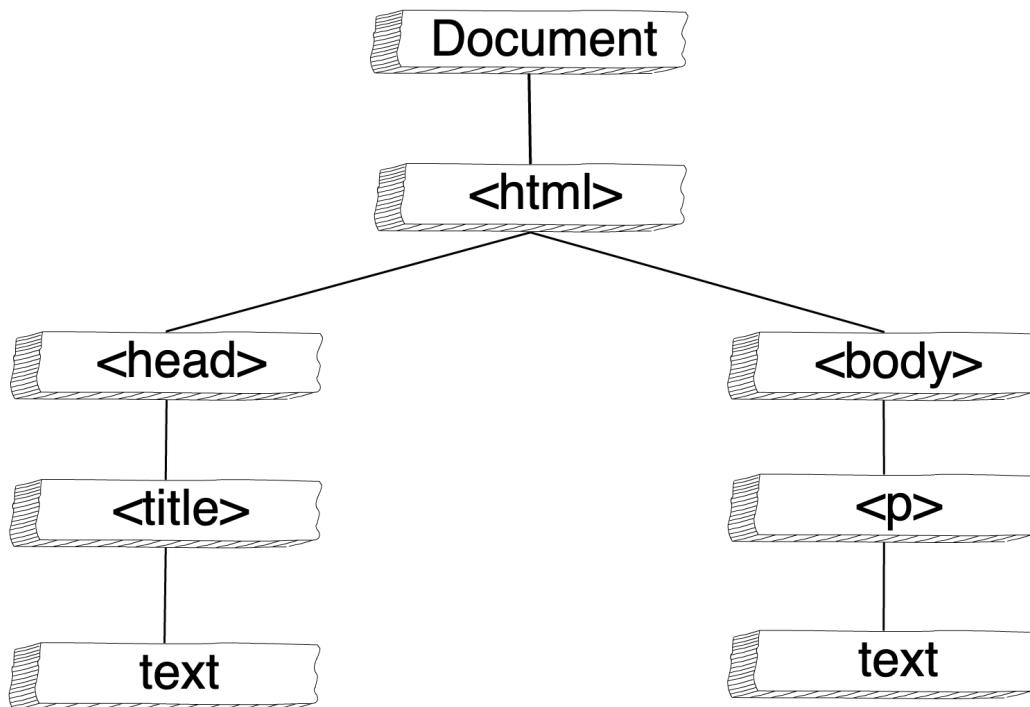
What is the difference between dynamic and static web pages?

Question

Does Node.js run in the browser environment?

23.2 The DOM

You may remember from earlier chapters that classes represent specific entities. The **Document Object Model (DOM)** is a set of objects that represent the browser and the documents that make up the web page. The DOM objects are instances of classes that model the structure of the browser, HTML document, HTML elements, element attributes, and CSS. The below figure depicts the parent-child relationships between the DOM objects that make up a web page.



23.2.1 Global DOM Variables

To utilize the DOM in JavaScript, we need to use the DOM global variables. In the next section, we will learn more about the DOM global variables, including their type. For now, let's get used to the idea of using JavaScript to interact with the DOM.

To start, we are going to use the `window` and `document` global variables. As mentioned above, we will go into more detail on these variables and what they are later.

Example

Here, the `window` and `document` variables are used to print information about the web page to the browser's console.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Using DOM Variables  title
5         script
6           "the page title:"
7           "the protocol:"
8         script
9     head
10    body
11      contents
12    body
13  html
```

Console Output

```
the page title: Using DOM Variables
the protocol: https:
```

23.2.2 Dynamic Web Page Using the DOM

The DOM plays a key part in making web pages dynamic. Since the DOM is a JavaScript representation of the web page, you can use JavaScript to alter the DOM and consequently, the web page. The browser will re-render the web page anytime changes are made via the DOM.

Note

Rendering is not the same action as loading.

In order to add or edit HTML elements with code, we need to be able to access them. The method `document.getElementById` will search for a matching element and return a reference to it. We will go into more detail on how this method works in the next section.

Example

We can use `document.getElementById` and `element.append` to add text to a `<p>` tag.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Add content using DOM  title
5     head
6     body
7       p id "main-text" Words about things...  p
8         script
9           let
```

(continues on next page)

(continued from previous page)

```
10      "More words about things"  
11  
12      script  
13  body  
14  html
```

Console Output

```
Words about things... More words about things
```

23.2.3 Where to Put the <script>

In the previous example, notice the `<script>` tag is placed below the `<p>` tag in the HTML document. HTML documents are executed top down. Therefore, a `<script>` tag must come after any other elements that will be affected by the code inside the `<script>`. Later in the chapter, we will learn about another way to handle this.

23.2.4 Check Your Understanding

Question

What do the DOM objects represent?

- a. Word documents you have downloaded
 - b. Directives of memory
 - c. The browser window, HTML document, and the elements
-

Question

What is the value of `p.innerHTML`?

```
1 p id "demo-text" Hello friend p  
2 script  
3   let  
4     "demo-text"  
5   script
```

23.3 More DOM Methods and Properties

The following sections are a summary of some DOM classes, methods, and properties. A more complete list can be found in the reference links below. You do NOT need to memorize everything on these reference pages. We are providing them to you as a guide for your future studies of the DOM.

1. [W3Schools DOM reference](#)
2. [MDN DOM reference](#)

23.3.1 Window

The global variable `window` is an instance of the `Window` class. The `Window` class represents the browser window. In the case of multi-tabbed browsers, the global `window` variable represents the specific tab in which the JavaScript code is running.

Table 1: Window Properties and Methods

Method or Property	Syntax	Description
<code>alert</code>	<code>window.alert("String message")</code>	Displays a dialog box with a message and an “ok” button to close the box.
<code>confirm</code>	<code>window.confirm("String message")</code>	Displays a dialog box with a message and returns <code>true</code> if user clicks “ok” and <code>false</code> if user clicks “cancel”.
<code>location</code>	<code>window.location</code>	Object that represents and alters the web address of the window or tab.
<code>console</code>	<code>window.console</code>	Represents the debugging console. Most common and basic use is <code>window.console.log()</code> .

Note

When using JavaScript in the browser environment, methods and properties defined on the `Window` class are exposed as global functions and variables. An example of this is `window.console.log()` is accessible by referencing `console.log()` directly.

23.3.2 Document

The global `document` variable is an instance of the `Document` class. The `Document` class represents the HTML web page that is read and displayed by the browser. The `Document` class provides properties and methods to find, add, remove, and alter HTML elements inside on the web page.

Table 2: Document Properties and Methods

Method or Property	Syntax	Description
<code>title</code>	<code>document.title</code>	Read or set the title of the document.
<code>getElementById</code>	<code>document.getElementById("example-id")</code>	Returns a reference to the element that's <code>id</code> attribute matches the given string value.
<code>querySelector</code>	<code>document.querySelector("css selector")</code>	Returns the first element that matches the given CSS selector.
<code>querySelectorAll</code>	<code>document.querySelectorAll("css selector")</code>	Returns a list of elements that match the given CSS selector.

Note

`querySelector` and `querySelectorAll` use the CSS selector pattern to find matching elements. The pattern passed in must be a valid CSS selector. Elements will be found and returned the same way that elements are selected

to have CSS rules applied.

23.3.3 Element

HTML documents are made up of a tree of elements. The `Element` class represents an HTML element.

Table 3: Element Properties and Methods

Method or Property	Syntax	Description
<code>getAttribute</code>	<code>element.getAttribute("id")</code>	Returns the value of the attribute.
<code>setAttribute</code>	<code>element.setAttribute("id", "string-value")</code>	Sets the attribute to the given value.
<code>style</code>	<code>element.style.color</code>	Object that allows reading and setting <i>INLINE</i> CSS properties.
<code>innerHTML</code>	<code>element.innerHTML</code>	Reads or sets the HTML inside an element.

23.3.4 Check Your Understanding

Question

What value will `response` have if the user clicks *Cancel*?

```
let response = "String message"
```

Question

Which of these are TRUE about selecting DOM elements?

- a. You can select elements by *CSS class name*
 - b. You can select elements by *id attribute* value
 - c. You can select elements by *tag name*
 - d. All of the above
-

Question

What is the difference between the `document` and `window` variables?

23.4 Events

Have you ever thought about how programs respond to interactions from users and other programs? **Events** are code representations of these interactions that need to be responded to.

In programming, events are triggered and then handled.

Events in programming are triggered and handled. **Triggering** an event is the act of causing an event to be sent. **Handling** an event is receiving the event and performing an action in response.

23.4.1 JavaScript and Events

JavaScript is an event-driven programming language. **Event-driven** is a programming pattern where the flow of the program is determined by a series of events. JavaScript uses events to handle user interaction and make web pages dynamic. JavaScript also uses events to know when the state of the web page components change.

23.4.2 DOM Events

Running JavaScript in the browser requires a specific set of events that relate to loading, styling, and displaying HTML elements. Objects in the DOM have event handling built right into them.

Some elements, such as `a`, have default functionality that handles certain events. An example of default event handling is when a user clicks on an `<a>` tag, the browser will navigate to the address in the `href` attribute.

Note

The DOM defines *numerous* events. Each element type does NOT support every event type. The kinds of events that each element supports relate to how the element is used.

23.4.3 Handling Events

Feature-rich web applications rely on more than the default event handling provided by the DOM. We can add custom interactivity with the users by attaching event handlers to HTML elements and then writing the event handler code.

To write a handler, you need to tell the browser what to do when a certain event happens. DOM elements use the `on event` naming convention when declaring event handlers.

The first way we will handle events is to declare the event handler in HTML, this is often referred to as an **inline event handler**. For example, when defining what happens when a `button` element is clicked, the `onclick` attribute is used. This naming convention can be read as: *On click of the button, print a message to the console.*

Example

```
1 <!DOCTYPE html>
2   html
3     head
4       title Button click handler  title
5     head
6     body
7       button onclick "console.log('you rang...');" Ring Bell  button
8     body
9   html
```

Console Output (if button is clicked)

```
you rang...
```

Tip

Notice the use of single quotes around 'you rang...'. When declaring the value of an attribute to be a string, you must use single quotes ' inside the double quotes ".

Note

button elements represent a clickable entity. button elements have default *click* handling behavior related to form elements. That we will get into in a later chapter. For now, we will be defining the *click* handler behavior.

Any JavaScript function can be used as the event handler. That means any defined functions can be used. Because programmers can write functions to do whatever their hearts desire, defined functions as event handlers allow for more functionality to occur when an event is handled.

Example

A function youRang () is defined and used as the event handler for when the button is clicked.

```
1  <!DOCTYPE html>
2  html
3  head
4      title Button click handler title
5      script
6          function
7              "main-text"
8                  "you rang..."
9
10     script
11     head
12     body
13         h1 demo header h1
14         p id "main-text" class "orange" style "font-weight: bold;" 
15             a bunch of really valuable text...
16         p
17         button onclick "youRang();" Ring Bell button
18     body
19     html
```

Result (if button is clicked)

```
effect on page: adds "you rang..." to <p>
output in console: you rang...
```

Warning

When defining handlers via HTML, be very careful to type the function name correctly. If the function name is incorrect, the event will not be handled. No warning is given, the event is silently ignored.

23.4.4 Check Your Understanding

Question

What does an *event* represent in the browser JavaScript environment?

Question

Why is JavaScript considered an *event-driven* language?

Question

Receiving an event and responding to it is known as?

- a. Holding an event
 - b. Having an event
 - c. Handling an event
-

23.5 Event Listeners

Using inline event handling is a good way to get started handling events. A second way to handle events uses the DOM objects and methods. Remember, the DOM is an object representation of the entire web page. The DOM allows us to use JavaScript to configure our event handlers. The event handling declaration will no longer be in the HTML element attribute, but will instead be inside `<script>` tags or in an external JavaScript file.

23.5.1 Add Event Handlers in JavaScript

Before we add event handlers in JavaScript, we need to learn a new vocabulary term related to events in programming. A **listener** is another name for an event handler. The term listener refers to the code *listening* for the event to occur. If the code *hears* the event, then the event is handled.

`addEventListener` is used to add an event handler, aka *listener*. `addEventListener` is a method available on instances of `Window`, `Document`, and `Element` classes.

```
"eventName"
```

`anElement` is a reference to a DOM element object. `"eventName"` is the name of an event that the variable `anElement` supports. `aFunction` is a reference to a function. To start, we are going to use a *named function*.

Example

We want to set the named function `youRang` as the *click* handler for the `button` element. Notice that the value passed in as the event name is `"click"` instead of `"onclick"`.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Use addEventListener  title
5     head
6   body
7     p id "main-text" class "orange" style "font-weight: bold;" 
8       a bunch of really valuable text...
9     p
```

(continues on next page)

(continued from previous page)

```

10   button id "ring-button" Ring Bell button
11   script
12     function
13       "main-text"           "you rang..."
14       "you rang..."
15
16   // Obtain a reference to the button element
17   let button = document.getElementById("ring-button")
18   // Set named function youRang as the click event handler
19   button.addEventListener("click", youRang)
20
21   script
22   body
23   html

```

Result (if button is clicked)

```
affect on page: adds "you rang..." to <p>
output in console: you rang...
```

Warning

Be sure to use the correct event name when declaring the event name. An error will NOT be thrown if an invalid event name is given.

Note

This chapter uses DOM methods to add event handlers. When searching online, you may find examples using jQuery to add event handlers, which look like `.on("click", ...)` or `.click(...)`. **jQuery** is a JavaScript library designed to simplify working with the DOM. jQuery's popularity has declined as the DOM itself has gained features and improved usability.

The second parameter of `addEventListener` is a function. Remember there are many ways to declare a function in JavaScript. So far, we have passed in named functions as the event handler. `addEventListener` will accept any valid function as the event handler. It's possible, and quite common, to pass in an *anonymous function* as the event handler.

```

1   "eventName"  function
2   // function body of anonymous function
3   // this function will be executed when the event is triggered
4

```

23.5.2 Event Details

A benefit of using `addEventListener` is that an *event* parameter is passed as the parameter to the event handler function. This event is an object instance of the `Event` class, which defines methods and properties related to events.

```

1   "eventName"  function
2   "event type"
3   "event target"
4

```

`event.type` is a string name of the event.

`event.target` is an element object that was the target of the event.

Try It!

Above, we saw how we could use `addEventListener` to add the function `youRang()` as the event handler for the Ring Bell button.

Using `addEventListener`, could you add the function `greetFriends()` as the event handler for the Greet Friends button?

[Try it at repl.it](#)

23.5.3 Event Bubbling

Remember that the DOM is a tree of elements with an `<html>` element at the root. The tree structure of an html page is made of elements inside of elements. That layering effect can cause some events, like `click`, to be triggered on a series of elements. **Bubbling** refers to an event being propagated to ancestor elements, when an event is triggered on an element that has parent elements. Events are triggered first on the element that is most closely affected by the event.

Example

We can add a `click` handler to a `<button>`, a `<div>`, and the `<html>` element via the `document` global variable.

```
1  <!DOCTYPE html>
2  html
3      head
4          title Event Bubbling  title
5          style
6
7              padding  20px
8              border  1px solid black
9              background-color darkcyan
10
11         style
12     head
13     body
14         div id "toolbar"
15             button id "ring-button" Ring Bell  button
16             div
17             script
18                 let
19                     "ring-button"
20                         "click"  function
21                         "button clicked"
22
23                         "toolbar"          "click"  function
24                         "toolbar clicked"
25
26                         "click"  function
27                         "document clicked"
28
29             script
30     body
31     html
```

Console Output (if button is clicked)

```
button clicked  
toolbar clicked  
document clicked
```

In some cases, you may want to stop events from bubbling up. We can use `event.stopPropagation()` to stop events from being sent to ancestor elements. Handlers for parent elements will not be triggered if a child element calls `event.stopPropagation()`.

```
1           "click"  function  
2         "button clicked"  
3  
4
```

Try It!

With the HTML above, what happens when you click in the green?

After you see the result, try adding `stopPropagation()` to the button click handler and seeing what happens when you click the button.

[Try it at repl.it](#)

23.5.4 Check Your Understanding

Question

Do these code snippets have the same effect? `button.addEventListener("click", youRang)` and `<button onclick="youRang();">`

Question

Can `click` events be prevented from bubbling up to ancestor element(s)?

Question

What is passed as the `argument` to the event handler function?

23.6 Event Types

DOM and JavaScript can handle numerous event types. We will discuss a few different types of events here. As you continue your studies of the DOM and events, you may find these two reference links helpful.

1. [W3Schools Event reference](#)
2. [MDN Event reference](#)

23.6.1 Load Event

The DOM includes the **load event**, which is triggered when the window, elements, and resources have been *loaded* by the browser. Why is it important to know when things have loaded? Remember you can't interact with HTML elements in JavaScript unless they have been loaded into the DOM.

Previously, we were moving the `<script>` element *below* any HTML elements that we needed to reference in the DOM. Using the *load event* on the global variable `window` is an alternative to `<script>` placement. When the *load event* has triggered on the `window` as a whole, we can know that all the elements are ready to be used.

Example

A `<script>` tag is in `<header>` and all DOM code is inside *load* event handler.

```
1  <!DOCTYPE html>
2  html
3  head
4      title Window Load Event  title
5      script
6          // add load event handler to window
7          "load"  function
8              // put DOM code here to ensure elements have been loaded
9              'window loaded'
10
11     let
12         "ring-button"
13         "click"  function
14         "ring ring"
15
16     let
17         "knock-button"
18         "click"  function
19         "knock knock"
20
21     script
22     head
23     body
24         div id "toolbar"
25             button id "ring-button" Ring Bell  button
26             button id "knock-button" Knock on Door  button
27         div
28     body
29     html
```

Console Output (if “Knock on Door” button is clicked)

```
window loaded
knock knock
```

23.6.2 Mouseover Event

There are many mouse related DOM events. We have already used the `click` event. Another example of a mouse event is the **mouseover** event, which is triggered when the mouse pointer enters an element.

Example

We can use *mouseover* event to add a ">" to the `innerHTML` of the element that the mouse pointer has been moved over.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Mouseover Event  title
5       script
6           "load"  function
7             let
8               "lane-list"
9               "mouseover"  function
10              let
11                "gt"
12                "target"
13
14      script
15    head
16  body
17    Mouseover Race
18    ul id "lane-list"
19      li Lane 1  li
20      li Lane 2  li
21      li Lane 3  li
22    ul
23  body
24  html
```

Example HTML Output (if the mouse is moved over elements in the list)

```
Mouseover Race
```

```
  Lane 1>>>>>
  Lane 2>>>>>>>>>
  Lane 3>>>>>>
```

23.6.3 Check Your Understanding

Question

What error happens when you try to access an element that has not been loaded into the DOM?

Question

What is *true* when the *load* event is triggered on `window`?

- a. The console is clear.
 - b. All elements and resources have been loaded by the browser.
 - c. Your files have finished uploading.
-

23.7 Exercises: The DOM and Events

Time to make a flight simulator for your fellow astronauts! The provided HTML and JavaScript files can be used for all of the exercises. For each exercise, the requirements and desired effect of the events is listed.

[Repl.it with starter code](#)

1. When the “Take off” button is clicked, the text “The shuttle is on the ground” changes to “Houston, we have lift off!”. The “Take off” button has an `id` of “liftoff”.
2. When the user’s mouse goes over the “Abort Mission” button, the button’s background turns red. The “Abort Mission” button has an `id` of “abortMission”.
3. When the user clicks the “Abort Mission” button, make a confirmation window that says “Are you sure you want to abort the mission?”.

23.8 Studio: The DOM and Events

Now that we can build a basic flight simulator, we want to add more controls for the staff at our space station. The HTML, CSS, and JavaScript files are provided. For each event, the requirements and desired effect is listed.

[Repl.it with starter code](#)

1. Use the window `load` event to ensure all elements have loaded before attaching event handlers.
2. When the “Take off” button is clicked, the following should happen:
 1. A window confirm should let the user know “Confirm that the shuttle is ready for takeoff.” If the shuttle is ready for liftoff, then add steps 2-4.
 2. The flight status should change to “Shuttle in flight”.
 3. The background color of the shuttle flight screen (`id = "shuttleBackground"`) should change from green to blue.
 4. The shuttle height should increase by 10,000 miles.
3. When the “Land” button is clicked, the following should happen:
 1. A window alert should let the user know “The shuttle is landing. Landing gear engaged.”
 2. The flight status should change to “The shuttle has landed”.
 3. The background color of the shuttle flight screen should change from blue to green.
 4. The shuttle height should go down to 0.
4. When the “Abort Mission” button is clicked, the following should happen:
 1. A window confirm should let the user know “Confirm that you want to abort the mission.” If the user wants to abort the mission, then add steps 2-4.
 2. The flight status should change to “Mission aborted.”
 3. The background color of the shuttle flight screen should change from blue to green.
 4. The shuttle height should go do to 0.
5. When the “Up”, “Down”, “Right”, and “Left” buttons are clicked, the following should happen:
 1. The rocket image should move 10 px in the direction of the button that was clicked.
 2. If the “Up” or “Down” buttons were clicked, then the shuttle height should increase or decrease by 10,000 miles.

23.8.1 Bonus Mission

If you are done with the above and have some time left during class, there are a few problems that you can tackle for a bonus mission.

1. Keep the rocket from flying off of the background.
2. Return the rocket to its original position on the background when it has been landed or the mission was aborted.

HTTP: THE POSTAL SERVICE OF THE INTERNET

24.1 How the Internet Works

Most people use the Internet without fully understanding how it works. Without much trouble, they can open a browser, navigate to a site, and interact with it. They do not need to know precisely *how* the Internet works in order to use it.

For web developers, however, fundamental understanding of the flow of information across the Internet is essential.

24.1.1 Servers and Clients

The Internet uses the **client-server model**. A **server** is an application that provides resources—such as raw data, web pages, or images. A **client** is an application that requests resources from a server.

When navigating the web, the client is the web browser on your computer or smartphone. When you click on a link or type in an address and hit *Enter*, the client/browser makes a request to a server that sits in a building somewhere out in the world. The server receives the request, and sends a response back to the client. The client then displays the content of the response.

Fun Fact

In the client-server model, the server may sometimes be on the *same* computer as the client. This is often the case when a programmer is building a web application. The in-progress, development version of the application is on their laptop, as is their browser that they use to test the app.

24.1.2 Protocols

A **protocol** is a standard for communication between computers. Most web communication uses *three* protocols, in fact.

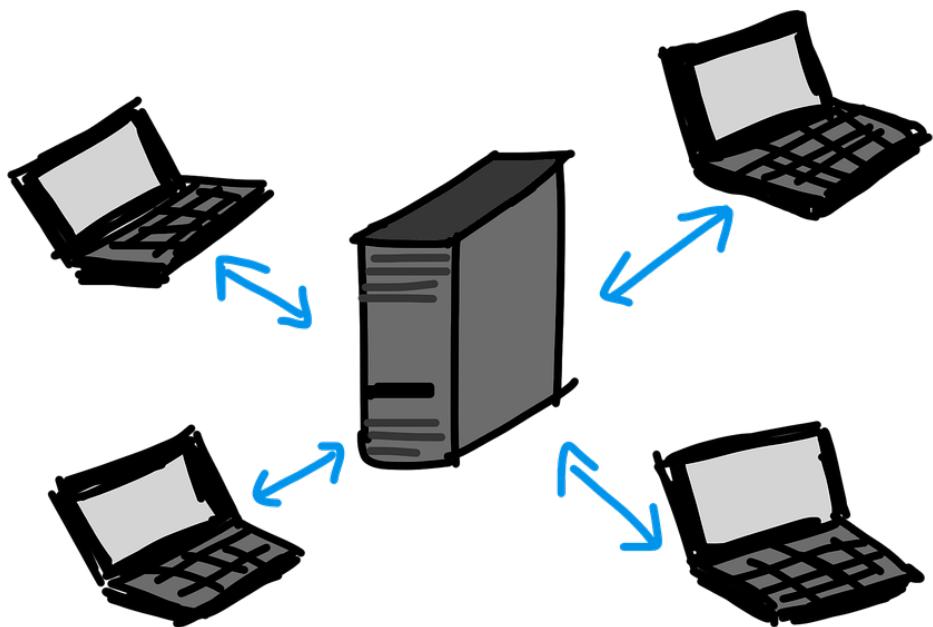


Fig. 1: Several clients communicating with a server.

Table 1: Common Web Protocols

Protocol	Full Name	Role
HTTP	Hypertext Transfer Protocol	High-level web communication for transferring files and information, including: <ul style="list-style-type: none"> HTML, CSS, and JavaScript files images and other media form submissions
TCP/IP	Transmission Control Protocol / Internet Protocol	Low-level web communication for transferring small chunks of raw data known as <i>packets</i>
DNS	Domain Name Service	Translates human-friendly names into server addresses

A thorough understanding of each of these protocols is well beyond the scope of this class. However, as a web developer it is important you have a general understanding of their roles. Each protocol has a different and critical job in enabling web communication.

HTTP

HTTP is the most important protocol for web developers to understand, which you may have guessed from the title of this chapter. It specifies how requests for common web data—such as HTML files or images—should be structured, as well as responses to such requests. The details of request and response message structure are the topic of the rest of this chapter.

HTTPS refers to the HTTP protocol used with a secure connection. A secure connection encrypts so that it can't be read while in-transit. The data is encrypted by the server/client before being transmitted, and decrypted once it is received by the client/server. The precise details of how such encryption works is beyond the scope of this course.

TCP/IP

TCP/IP is a low-level protocol that is quite complicated. For our purposes, it is important only to know that TCP/IP is the standard that allows *raw data* to get from one place to another on the Internet.

When a server sends a file back to a client, that file must physically be sent across a series of network components, including cables, routers, and switches. Files are broken down into *packets*—small chunks of a standard size—that are individually sent from one location to the next, until arriving at their final destination and being reassembled.

Fun Fact

You can think of the Internet as a “series of tubes.” This phrase was used by a U.S. Senator in 2006 and widely mocked. However, we think it's actually a reasonable analogy. TCP/IP allows data to be passed from one tube to another until reaching the final destination.

DNS

DNS is the address book of the Internet. It enables us to use readable and memorable names for servers, such as `www.launchcode.org` or `mail.google.com`. Such names are called **domain names**, and they function as aliases for the actual server addresses.

Every server on the internet has a numerical address known as an **IP address**. When a message is addressed using a domain name, the corresponding IP address must be determined before it can be sent.

Example

The IP addresses of `www.launchcode.org` and `mail.google.com` are `104.25.127.113` and `172.217.5.229`, respectively.

The sending computer will attempt to *resolve* the domain name by looking it up on a nameserver. A **nameserver** is a directory of domains and IP addresses, and there are thousands of them on the Internet. Most internet service providers (such as Charter or AT&T) provide DNS servers for their customers to use. Once the sending computer knows the IP address, it can send the request to the correct server.

Try It!

It's easy to look up the IP address of any domain name using freely-available tools.

Use the popular site [MX Toolbox](#) to look up the IP address of `help.launchcode.org`. Does this site live on the same server as `launchcode.org`?

Fun Fact

Every computer uses the special IP address `127.0.0.1` to refer to *itself*. This is known as the **loopback address**, and it often has the alias `localhost`. If you use the loopback address when making a request, the request will be sent to a service on the *same* machine as the client.

24.1.3 Web Addresses

When a client requests a resource from a server, it does so using a **uniform resource locator (URL)**. URLs are also called **web addresses**.

Examples

As a regular user of the Internet, you are already familiar with URLs like these:

- `https://www.launchcode.org`
 - `https://en.wikipedia.org/wiki/Client-server_model`
 - `https://duckduckgo.com/?q=javascript`
-

A URL encodes a lot of information about the request, including *what* is being requested and *where* the request should be sent. URLs are made up of several components, each of which plays a role in enabling both client and server to understand what is being requested.

We will generally work with URLs with this structure:

```
scheme://host:port/path?query_string
```

The five components of this URL are:

- Scheme
- Host

- Port (optional)
- Path (optional)
- Query String (optional)

A properly-formed URL must have these components in the *exact* order shown here. Only scheme and host are required.

Let's look at each of these in detail.

Scheme

The first portion of every URL specifies the **scheme**. Common schemes are `http`, `https`, `ftp`, `mailto`, and `file`. Most often, the scheme specifies the *protocol* to be used in making a request. For us, this will always be `http` or `https`. If left off, a web browser will insert the scheme `http/s` for you.

The scheme is *always* followed by `://`.

Host

The **host** portion of a URL specifies *where* the request should be sent. The host can be either an IP address, like `104.25.128.113`, or a domain name, like `www.launchcode.org`.

Port

Following the host is an optional **port** number. While the host determines the *server* that the request should be sent to, the port determines the specific *application* on the server that should handle the request. This is important because a single server may run several applications capable of handling requests.

Conventionally, a given type of application will always use the same port, though this is not a hard rule. For example, web servers typically use port 80 or 443, for regular and encrypted messages, respectively. On the other hand, MySQL databases typically use port 3306.

Example

Suppose a server at `mydomain.com` is running both a web server and MySQL database server on the standard ports. Requests to `mydomain.com:80` will be given to the web server, while requests to `mydomain:3306` will be given to the database server.

If a port number is not specified, then a default value based on the scheme is used. When using `http://` the default port is 80. When using `https://` the default port is 443.

Path

Following the domain and optional port is the **path**, which consists of a series of names separated by `/`. The path provides information that tells the server *what* is being requested. It can consist of a series of names, such as `/blog/entries/2018/`, or it can end with an explicit file name, such as `/blog/index.html`.

Example

A request to `https://www.launchcode.org/blog/` asks for the resource that lives at the path `/blog/` on the server `www.launchcode.org`. This resource happens to be the homepage of the LaunchCode blog.

A request to the (very long) URL below asks for the LaunchCode logo, which lives at the path /assets/dabomb-2080d6e...57f.svg (truncated here for space).

```
https://www.launchcode.org/assets/dabomb-
˓→2080d6e23ef41463553f203daaa15991fd4c812676d0b098243b4941fcf4b57f.svg
```

If a path is not specified, then the **root path** / is used. The root path typically refers to the home page for a given site.

Query String

Following the path is an optional **query string**, which begins with ? and contains a set of key-value pairs. Each pair is joined by = and is separated from the other pairs by &. For example, the query string of a [search on duckduckgo.com](#) looks like this:

```
?q=launchcode&atb=v167-5__&ia=web
```

This query string has *three* key-value pairs:

- q : launchcode
- atb : v167-5__
- ia : web

Notice that these pairs are separated by & in the query string.

While the path specifies *what* the request is asking for, the query string provides additional data that may be needed to fulfill the request. As an analogy, you can think of the path like a function name, and the query string as the function arguments.

24.1.4 Putting It All Together

We just covered a *lot* of information! While these nuts-and-bolts details are important, they aren't nearly as important as the high-level picture of how we access resources on the internet.

To tie these ideas together, watch these two videos on URLs and the Internet as a whole:

- [How Do URLs Work?](#)
- [How the Internet Works](#)

24.1.5 Check Your Understanding

Question

Which protocol is responsible for turning a name like `launchcode.org` into a server address?

Question

Why is this URL malformed?

```
https://launchcode.org?city=miami/lc101
```

1. It uses HTTPS when it should use HTTP.

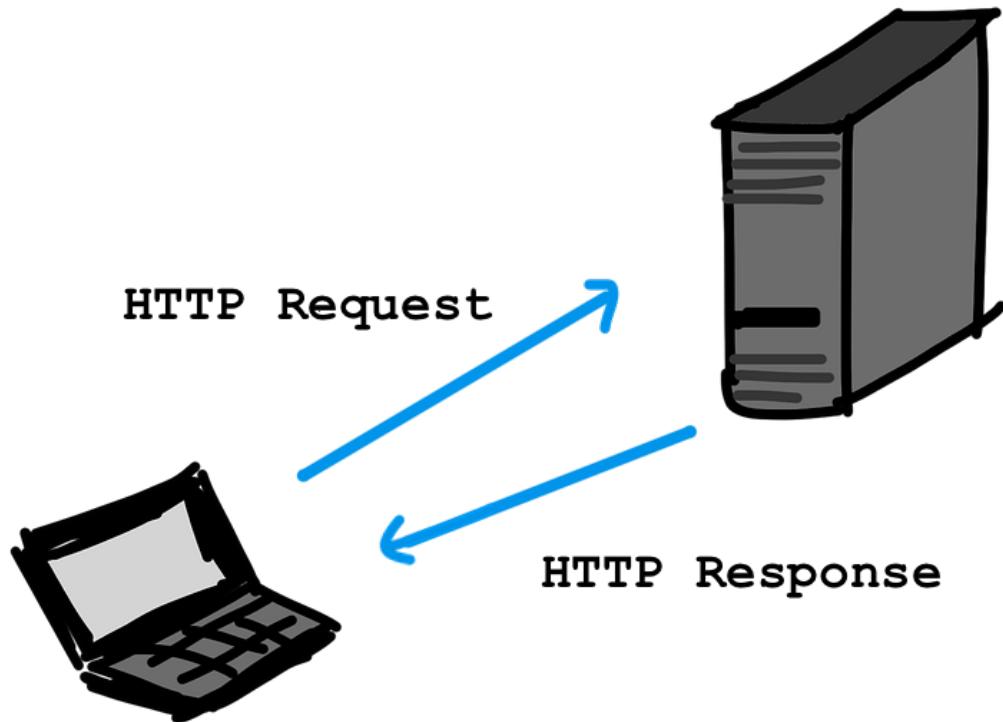
2. It doesn't contain a fragment.
 3. It doesn't contain a port.
 4. The query string comes before the path.
-

24.2 HTTP at a Glance

While web developers can often get away with a minimal understanding of TCP/IP and DNS, they must understand HTTP much more deeply. Before diving into the details of HTTP, let's gain a high-level understanding.

24.2.1 Requests and Responses

The fundamental units of HTTP are **requests** and **responses**. A client (usually a web browser) makes a request to a web server. Based on the details of the request, the server formulates and sends a response. The response is parsed and displayed by the browser.



As long as the server is available, *every* request receives a single response.

Requests contain several types of data, including:

- The URL being requested.
- The type of action the client is asking the browser to take.
- Metadata about the request, such as the type of browser making the request and the type(s) of data the client can accept in return.

- Optionally, a request message.

Responses include:

- The status of the response, including success or failure reasons.
- Metadata about the response, including the size and data format of the response message.
- Optionally, a response message.

24.2.2 The Postal Service of the Internet

HTTP can seem complicated, but it is actually very similar to a system that you are already familiar with: The United States Postal Service.

Suppose you want to mail a letter to your friend in Alaska, asking them their favorite cheese. To do so, you write your question on a piece of paper and enclose it in an envelope. On the envelope, you write your friend's address, along with your return address. Finally, you affix a stamp to the top-right corner.

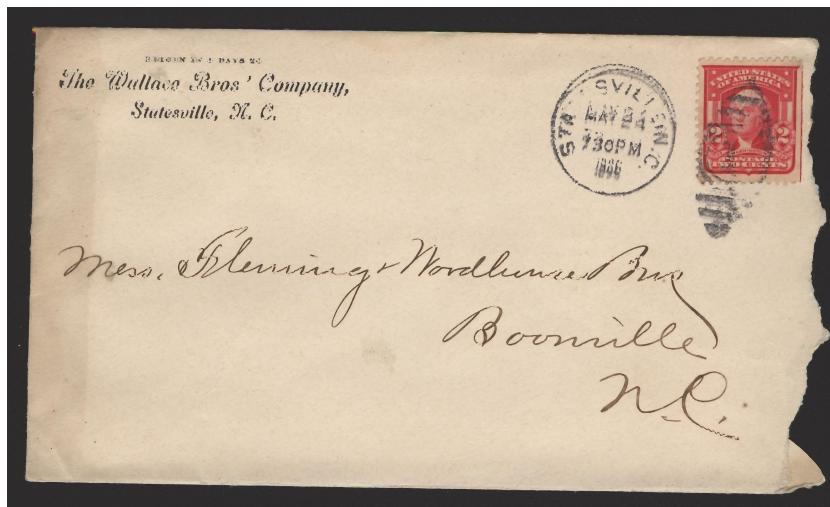


Fig. 2: Image is in the public domain

Each of these pieces of information is necessary for your letter to be delivered. When your letter enters the postal system, it will travel from one post office to another, via land, air, and maybe even sea. As long as you follow their rules, the postal service will get your letter to its destination.

This is very similar to how an HTTP request works. The letter is like a request message. The envelope contains the location and metadata needed for the letter to be delivered, just like an HTTP request specifies a URL and other metadata necessary for the request to reach the server and be processed.

When you drop the letter in your mailbox, you know it will be delivered since you followed the postal service's rules. When we make HTTP requests, we don't know *how* our request will get to the server, but as long as we properly structure a request, it *will* be delivered.

And just as your friend will respond with a letter telling you their favorite cheese (sharp white cheddar!), an HTTP request will result in a response from the server.

As we wade into the details of HTTP, keep this analogy in mind. It will help simplify the concepts and make them more concrete.

Question



Fig. 3: Both the postal service and the Internet deliver messages, as long as you follow their formatting rules. Images used with permission. L: via US Air Force, R: via Flickr user verkeorg

In your own words, explain the role of HTTP in enabling communication over the Internet.

Question

Answer true or false for each of the following statements.

1. A web server can send multiple responses for a single HTTP request.
 2. The postal service will deliver your HTTP requests, if you ask nicely.
 3. When creating an HTTP request, we must specify every network connection and server between our client and the server.
 4. The postal service is a good analogy for HTTP.
-

24.3 Requests

HTTP requests must conform to the structure outlined by the [World Wide Web Consortium \(W3C\)](#). We'll discuss the most important and most commonly-used aspects of HTTP request structure.

A generic HTTP request looks like this:

```
GET /blog/ HTTP/1.1
Host: www.launchcode.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:67.0) Gecko/20100101
  ↪Firefox/67.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0

Request Body
```

The structure has these components:

1. **Request line:** The first line is the request line. It specifies the **request method**, path, and HTTP version being used.
2. **Request headers:** From line 2 until the first blank line, **request headers** are included as a series of key-value pairs, one per line.
3. **Blank line:** This signifies the end of the request headers.
4. **Request body (Optional):** Below the blank line, the request body takes up the remainder of the HTTP request.

24.3.1 Request Methods

The request line is minimal. We have already discussed the path, which specifies the resource being requested. The first part of the request line, the request method, is new to us.

A **request method** specifies the type of action to be carried out on the requested resource. HTTP defines [8 methods](#), of which we will only need to use 2: GET and POST.

The GET Method

Using the `GET` method tells the server that we want to simply *retrieve* the resource. This is the most commonly used method. It is used for requests for HTML pages, CSS and JS files, and images. When you click on a link in a web page, you are triggering a `GET` request for the linked page.

`GET` requests generally do not have a body, since they are *asking* rather than *sending* for information.

Example

`GET` requests are usually used for:

- Loading a page after typing an address into the browser's address bar
 - Conducting a search via a search engine
 - Loading a page after clicking on a link
-

The POST Method

Using the `POST` method tells the server that we want to *create* new data on the server. As you will learn in the next chapter, `POST` is used when submitting a form.

`POST` requests usually have a body, which contains data that the server processes and stores in some fashion.

Example

Some common situations that use `POST` are:

- Signing into a web site
 - Sending an email via a web-based email client
 - Making an online purchase
-

24.3.2 Headers

There are quite a few request headers, but only a few will be useful to us.

Table 2: Common HTTP Request Headers

Header	Purpose	Example
Host	The domain name or IP address of the server the request should be sent to.	www.launchcode.org
User-Agent	Information about the client (usually a browser) making the request. The example is for a version of Firefox on a Mac.	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:67.0) Gecko/20100101 Firefox/67.0
Accept	The types of data that the client is willing to accept in the response body.	text/html, image/jpeg
Content-Type	The type of data included in the request body. Usually only used for <code>POST</code> requests.	application/json, application/xml

24.3.3 Body

The optional request body may contain any data whatsoever, though it often includes form data submitted via a POST request. For example, when signing into a web site, the request body will contain your username and password. We will later learn that it can contain other data formats such as XML and JSON.

As mentioned above, GET requests generally do *not* have a body.

24.3.4 Check Your Understanding

Question

Which request type was used to load this page (the one you are currently reading)?

Question

Visit [Wikipedia's article on HTTP request headers](#). Which request header is used to set cookies? (Cookies are small pieces of data related to your interaction with a web site.)

24.4 Responses

Each HTTP request that reaches a web server results in an HTTP response to the client.

A generic HTTP response looks like this:

```
HTTP/2.0 200 OK
Date: Wed, 22 May 2019 17:36:50 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 8050
Last-Modified: Wed, 22 May 2019 17:33:45 GMT

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<!--Rest of HTML page -->
</html>
```

The structure has these components:

1. **Status line:** The first line of the response is the **response line**, which contains status information about the response including the **response code**. In this example, the response code is 200, which indicates the request was fulfilled successfully.
2. **Response headers:** Below the response line are the **response headers**. Similar to request headers, these are key-value pairs that contain metadata about the response.
3. **Blank line:** This signifies the end of the response headers.
4. **Response body (Optional):** Below the blank line, the request body takes up the remainder of the HTTP response. This is usually HTML, CSS, JavaScript, etc.

24.4.1 Response Codes

HTTP **response codes** are standardized codes that servers use to convey the result of attempting to fulfill the client's request. They are always three-digit numbers that fall into one of five categories based on the first digit.

- 1xx (Informational): The request was received but processing has not finished
- 2xx (Successful): The request was valid and the server successfully responded
- 3xx (Redirection): The client should go elsewhere to access the requested resource
- 4xx (Client Error): There was a problem with the client's request
- 5xx (Server Error): The client's request was valid, but the server experienced an error when fulfilling it

Specific codes will have all 3 digits specified, such as 201, 302, or 404. Each specific code has a specific meaning. One of the most commonly experienced error codes is 404. You have likely encountered a message like this at some point:

Error response

Error code: 404

Message: File not found.

Error code explanation: `HttpStatus.NOT_FOUND` - Nothing matches the given URI.

A 404 response code indicates that the requested resource does not exist on the server. This can occur when, for example, you make a typo when typing a URL into the address bar. Referring back to our postal service analogy, a 404 is similar to receiving a letter marked "Return to Sender" because the addressee doesn't live there anymore.

We don't expect you to memorize all of the response codes, but you should be able to quickly recall the most common codes.

Table 3: Common HTTP Response Codes

Code	Description	Example
200	The requested resource exists and was successfully returned.	Visiting any existing web page on the Internet.
301	The requested resource has moved, and the client should look for it at the URL included in the <code>Location</code> header.	A site moves a page, but wants users with old links to be redirected to the page's new location.
404	The server received the request, but the requested resource does not exist on the server.	Requesting an image or HTML file that does not exist on the server.
500	The server experienced an error while fulfilling the request.	The server lost its database connection and cannot retrieve requested data.

24.4.2 Response Headers

There are quite a few response headers, but only a few will be useful to us.

Table 4: Common HTTP Response Headers

Header	Purpose	Example
Content-Type	The type of data included in the response body.	text/html, text/css, image/jpg
Content-Length	The size of the response body in bytes.	348
Location	The URL that the client should visit to find a relocated resource.	https://www.launchcode.org/new-blog/

24.4.3 Response Body

While requests often don't have a body, requests almost *always* have a body. The response body is where the data that a request asked for is located. It can contain HTML, CSS, JavaScript, or image data.

When a response is received by a browser, it is loaded into the browser's memory, with additional processing in some cases. For HTML files, the markup is rendered into a web page. For CSS files, the style rules are parsed and applied to the given HTML page.

24.4.4 Check Your Understanding

Question

A 404 response indicates that:

1. The server is offline.
 2. The user needs to log in.
 3. The requested resource does not exist.
 4. The server's database crashed.
-

Question

Visit [Wikipedia's article on HTTP response codes](#). Which response code is used to signify that the user must authenticate themselves (that is, log in) before viewing the given resource?

24.5 HTTP in the Browser

While we have covered all of the HTTP concepts you need to know at this point, it's worth spending some time explaining how web browsers submit requests and receive responses.

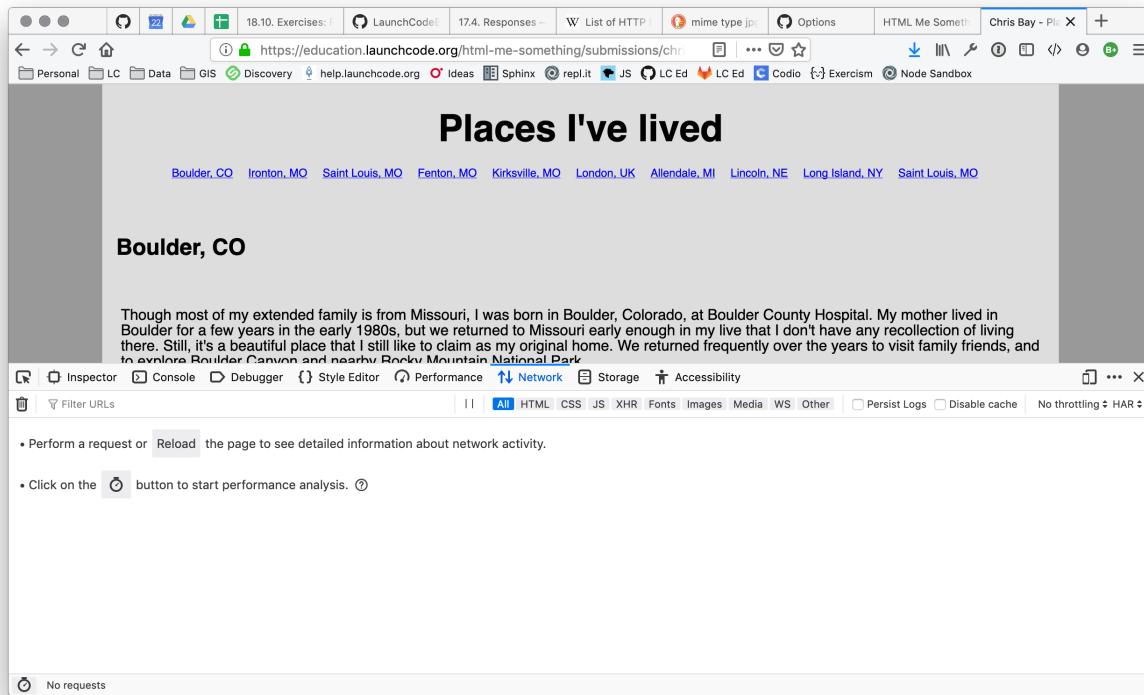
24.5.1 Viewing Requests and Response Using Developer Tools

Tip

This section requires you to use Firefox's developer tools. If you need a refresher or just a reference, visit [MDN](#).

Open a web browser and visit some web page, say, [our example from the HTML Me Something assignment](#). After the page loads, open your browser's developer tools and select the *Network* tab.

You'll see something like this:



The *Network* pane displays all HTTP requests and responses involved in loading a page. However, it only tracks and displays such data if it is open during the request. To see some data in this tab, refresh the page.

Now you'll see something like this:

Each entry within the pane represents a single HTTP request. A summary of the request is shown in a table format, including the resource requested, server name, response code, and more. Clicking on one of the entries shows more detailed information about the request.

On the right, we see additional request and response details, including response headers and (scrolling down) request headers. We can even view the response body by clicking on the *Response* label.

Try It!

Navigate to a different page with the *Network* pane open. Find the response code and `Content-Type` header for the first request shown in the pane.

24.5.2 Browser Flow

As you can see from using the *Network* pane, loading a single web page usually involves *several* HTTP requests. Each resource *within* the page is loaded in a separate request.

The screenshot shows a web browser window with the URL <https://education.launchcode.org/html-me-something/submissions/chrisbay>. The main content area displays a page titled "Places I've lived" with a heading "Boulder, CO". Below the heading is a paragraph of text. The browser's developer tools are open, specifically the Network tab, which shows a timeline of requests and a detailed HAR file below it.

Network Timeline:

- Request 1: GET /index.html (Status 200, 3.70 KB, 9.20 KB transferred, 66 ms)
- Request 2: GET /normalize.css (Status 200, 3.06 KB, 7.56 KB transferred, 97 ms)
- Request 3: GET /styles.css (Status 200, 1.08 KB, 976 B transferred, 58 ms)
- Request 4: GET /family.jpg (Status 404, 5.63 KB, 9.12 KB transferred, 31 ms)
- Request 5: GET /favicon.ico (Status 404, 0 KB, 9.12 KB transferred, 31 ms)

HAR File (HTTP Archive):

```

{
  "log": {
    "version": "1.2",
    "creator": "Google Chrome Developer Tools",
    "date": "2019-05-22T18:20:36Z",
    "http_version": "HTTP/2.0",
    "cookies": [
      {
        "name": "age",
        "value": "0"
      }
    ],
    "headers": [
      {
        "name": "cache-control",
        "value": "max-age=600"
      },
      {
        "name": "cf-ray",
        "value": "4dbce17d88f9ee5-ORD"
      },
      {
        "name": "date",
        "value": "Wed, 22 May 2019 18:20:36 GMT"
      },
      {
        "name": "etag",
        "value": "W/\"5a25c57c-24d0\""
      }
    ],
    "params": [],
    "responses": [
      {
        "url": "https://education.launchcode.org/html-me-something/submissions/chrisbay",
        "status": 304,
        "method": "GET",
        "headers": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": null
      }
    ],
    "resources": [
      {
        "url": "https://education.launchcode.org/html-me-something/submissions/chrisbay",
        "status": 304,
        "method": "GET",
        "headers": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": null
      },
      {
        "url": "https://education.launchcode.org/index.html",
        "status": 200,
        "method": "GET",
        "headers": [
          {
            "name": "Content-Type",
            "value": "text/html; charset=UTF-8"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": "The page content is here."
      },
      {
        "url": "https://education.launchcode.org/normalize.css",
        "status": 200,
        "method": "GET",
        "headers": [
          {
            "name": "Content-Type",
            "value": "text/css; charset=UTF-8"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": "The page content is here."
      },
      {
        "url": "https://education.launchcode.org/styles.css",
        "status": 200,
        "method": "GET",
        "headers": [
          {
            "name": "Content-Type",
            "value": "text/css; charset=UTF-8"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": "The page content is here."
      },
      {
        "url": "https://education.launchcode.org/family.jpg",
        "status": 404,
        "method": "GET",
        "headers": [
          {
            "name": "Content-Type",
            "value": "image/jpeg"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": "The page content is here."
      },
      {
        "url": "https://education.launchcode.org/favicon.ico",
        "status": 404,
        "method": "GET",
        "headers": [
          {
            "name": "Content-Type",
            "value": "image/x-icon"
          }
        ],
        "cookies": [
          {
            "name": "age",
            "value": "0"
          }
        ],
        "body": "The page content is here."
      }
    ]
  }
}

```

Let's examine the flow of loading a page. We'll consider the case of an HTML page with CSS, JavaScript, and images, loaded via a GET request.

1. Browser requests a page from the server.
2. Browser receives the HTML page and parses it.
3. For *each* image, external CSS file, and external JavaScript file the browser issues a *new* HTTP request for the given file.
4. As additional responses are received, the browser processes the data or media and updates the page.

This process explains why you will sometimes load a web page, only to see an image on that page load a few seconds later. In such situations, the HTTP request fetching the image takes substantially more time, making it noticeable.

24.5.3 Check Your Understanding

Question

For the first screenshot on this page, answer these questions:

1. What is its file name?
 2. How large is it?
 3. When was the file last modified?
-

CHAPTER
TWENTYFIVE

USER INPUT WITH FORMS

25.1 Forms

As a user of the web, you know that web pages both display and accept data. In this chapter we are going to learn more about how web pages handle data input using HTML forms. An HTML **form** is used to accept input from the user and send that data to the server.

25.1.1 Create a Form

To declare a form in HTML use the `<form>` tag with open and closing tags. This form element will serve as container for various types of other elements that are designed to capture input from the user.

```
1 html
2   head
3     title Form Example  title
4   head
5   body
6     <!-- empty form -->
7     form  form
8   body
9   html
```

An empty `<form></form>` will not appear on a web page until inputs have been added inside of it. Below we have added one basic `<input>` tag.

```
1 html
2   head
3     title Form Example  title
4   head
5   body
6     form
7       input type "text"
8     form
9   body
10  html
```

25.1.2 Input Element

The `input` element is used to add interactive fields, which allow the user to enter data. `input` elements have two very important attributes: *name* and *type*.

- The `name` attribute is used to identify the input's value when the data is submitted
- The `type` attribute defines which type of value of the input represents

```
input type "text" name "username"
```

Note

Notice that `<input type="text">` tags are self closing. **Self-closing** tags are *single* tags with `/>` at the end.

Warning

Values are NOT submitted for an `<input>` unless it has a `name` attribute.

25.1.3 Labels

Forms normally contain more than one input. `<label>` tags are used to provide a textual label, which informs the user of the purpose of the field. The simplest usage of `<label>` tags is to *wrap* them around `<input>` tags.

```
1 html
2   head
3     title Form Example title
4   head
5   body
6     form
7       label Username input type "text" name "username" label
8       label Team Name input type "text" name "team" label
9     form
10    body
11  html
```



A second way to relate a `<label>` tag to an `<input>` is to use the `id` attribute of `input` and the `for` attribute of `label`. The two are related by setting `for` in `<label for="username">` equal to the `id` of `<input id="username">`, these two attributes must be EQUAL. When `for` is used, the `<input>` does NOT have to be inside of the `<label>`.

```
1 label for "username" Username label
2   input id "username" name "username" type "text"
```

What happens when a `<label>` is clicked? The answer depends on what the `<label>` is associated to.

For `text` inputs, when the label is clicked, then the input gains *focus*. An element with **focus** is currently selected by the browser and ready to receive input.

Example

Click on the label text to the associated text input element gain focus.

```
1  div
2    label for "username" Username label
3    input id "username" name "username" type "text"
4  div
```

For *non-text* inputs, when the label is clicked, a value is selected. This behavior can be seen with `radio` and `checkbox` elements which we will learn more about soon.

Example

Click on the label text to the associated checkbox input element gain focus.

```
1  div
2    label Subscribe to Newsletter
3      input type "checkbox" name "newsletter"
4    label
5  div
```

25.1.4 Value Attribute

The `value` attribute of an `<input>` tag can be used to set the default value. If the `value` attribute is declared, then the browser will show that value in the input. The user can choose to update the value by typing in the input box.

Example

Input with default value of JavaScript.

```
div label Language input name "language" type "text" value "JavaScript" label
  ↵div
```

25.1.5 Check Your Understanding

Question

What is a self-closing tag?

Question

What is the purpose of the `name` attribute for `input` elements?

Question

Which `input` attribute sets the default value?

25.2 Form Submission

Forms collect data input by the user. As we learned in the previous chapter, communication on the web occurs via a series of HTTP requests and responses. A **form submission** is an HTTP request sent to the server containing the values in a form.

25.2.1 Trigger Form Submission

A form submission is triggered by clicking a button inside the form. A submit button can be an `input` element with `type=submit` or a `button` element. Both button types are in the below example.

```
1  form
2    label Username  input type "text" name "username"  label
3    <!-- clicking either of these will cause a form submission -->
4    input type "submit"
5    button Submit  button
6  form
```

When a form is submitted, an HTTP request is sent to the location set in the `action` attribute of the `<form>` tag.

If the `action` attribute is not present or is empty, then the form will submit to the URL of the current page.

Try It!

Open [this form](#) in a browser. Type values into the inputs, click the Submit button, and notice what happens to the address bar.

```
1  html
2    head
3      title Form Example  title
4      style
5        body  padding  25px
6        style
7      head
8    body
9      form action ""
10     label Username  input type "text" name "username"  label
11     label Team Name  input type "text" name "team"  label
12     button Submit  button
13   form
14   body
15  html
```

Output



Username

Team Name

Output After Submitted

The screenshot shows a browser window with a simple form. At the top, there are standard navigation icons (back, forward, refresh, home). The address bar displays a URL with a lock icon and the text "ault--launchcode.repl.co/?username=salina&team=Space+Coders". Below the address bar is the form itself, which consists of two text input fields labeled "Username" and "Team Name", and a single "Submit" button.

Run it.

Notice in the above example that the browser address has changed to:

```
https://form-default--launchcode.repl.co/?username=salina&team=Space+Coders
```

The web address is the same as the form we loaded, but now includes a query parameter for *every* input, with a name, in the form. These parameters are known as the query string parameters. The form values are submitted via the query string because the default submission type for forms is GET. In the next section will soon learn how to submit form data via POST.

Note

Since spaces are not allowed in URLs, the browser replaces them with +.

Key-value Pairs

When a form is submitted a key-value pair is created for each named input. The keys are the values of the name attributes, and they are paired with the content of the value attributes.

Form with two named inputs:

```
form action ""
  label Username  input type "text" name "username"  label
  label Team Name  input type "text" name "team"    label
  button Submit  button
form
```

When this form is submitted with the values from the previous example, the query string looks like this:

```
username=salina&team=Space+Coders
```

25.2.2 Check Your Understanding

Question

What must be added to a form to enable submission?

Question

By *default*, are HTTP forms submitted with GET or POST?

Question

When a form is submitted, where does the data for the key-value pairs come from?

25.3 POST Form Submission

25.3.1 Form Submission Using POST

Instead of using GET and query parameters to submit form data, we can use POST. To submit a form using a POST request, set the form's method attribute to "POST". Form data submitted via POST will be submitted in the body of the HTTP request. Data submitted by GET requests is less secure than POST because GET request URLs and the query parameters are cached and logged, possibly leaking sensitive data.

Example

Form with method="POST"

```
form action "" method "POST"
  label Username  input type "text" name "username"  label
  label Team Name  input type "text" name "team"    label
  button Submit  button
form
```

25.3.2 Send Form Submission to a Server

The action and method attributes allows us to choose where the form request will be sent and what type of request will be sent. How do we configure what happens in response to a form submission?

Form handlers are web server actions that receive, inspect, and process requests. They then send a response to the client. For this unit we are going to use form handlers that have already been created for us.

Example

When submitted this form will send a POST request to the form handler defined by the action attribute.

```
1  form action "https://handlers.education.launchcode.org/request-parrot" method "POST"
2    label Username  input type "text" name "username"  label
3    label Team Name  input type "text" name "team"    label
4    button Submit  button
5  form
```

repl.it

Try It!

1. Open example form that uses POST in a browser.

2. Open the network tab of the developer tools
3. Check “Persist Logs” in the network tab

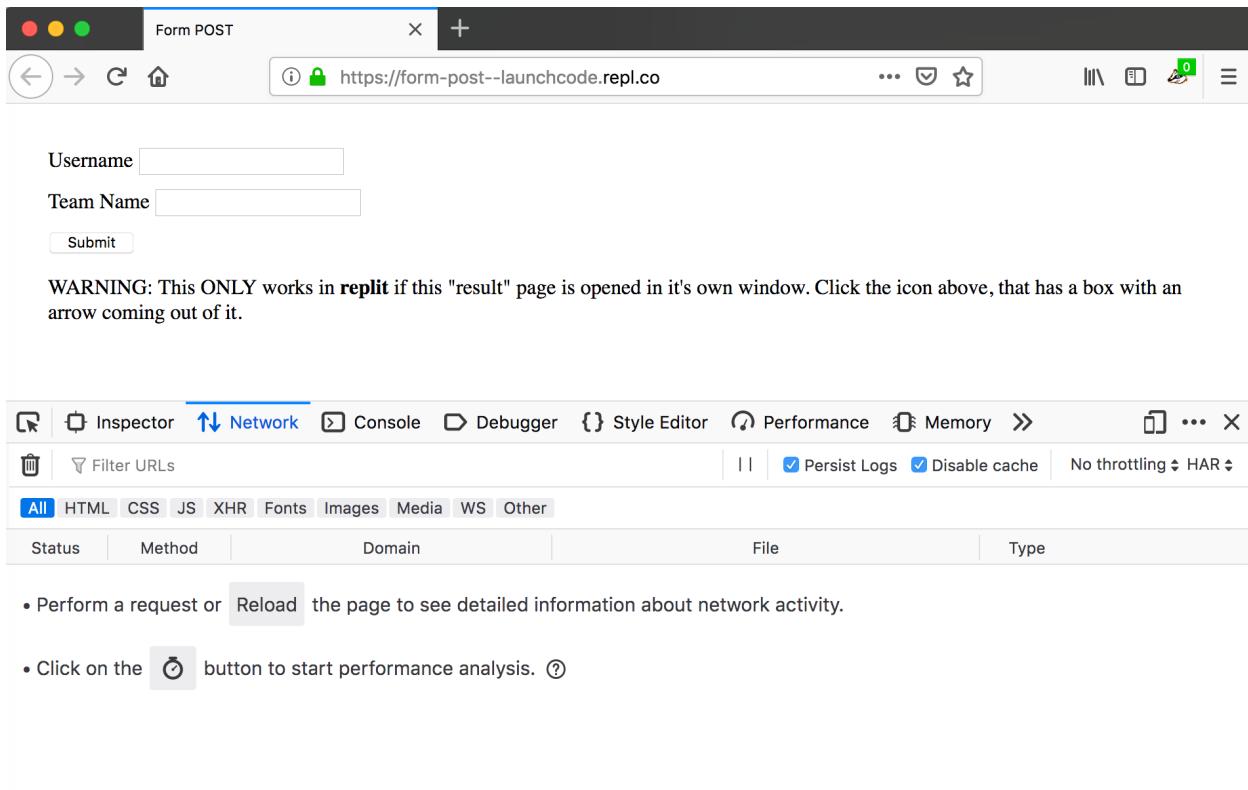


Fig. 1: Firefox browser with form loaded, network tab open, and Persist logs checked

4. Enter data into the inputs
 - Type `tracking` into Username input
 - Type `Requests` into Team Name input
 5. Click Submit button
 6. Inspect the data sent in the POST request
-

Warning

Using POST for form submissions adds a very low level of security. Using **HTTPS** instead of HTTP adds a higher level of security. Configuring HTTPS is beyond the scope of this class.

25.3.3 Check Your Understanding

Question

What attribute on `<form>` determines if the form is submitted with GET or POST?

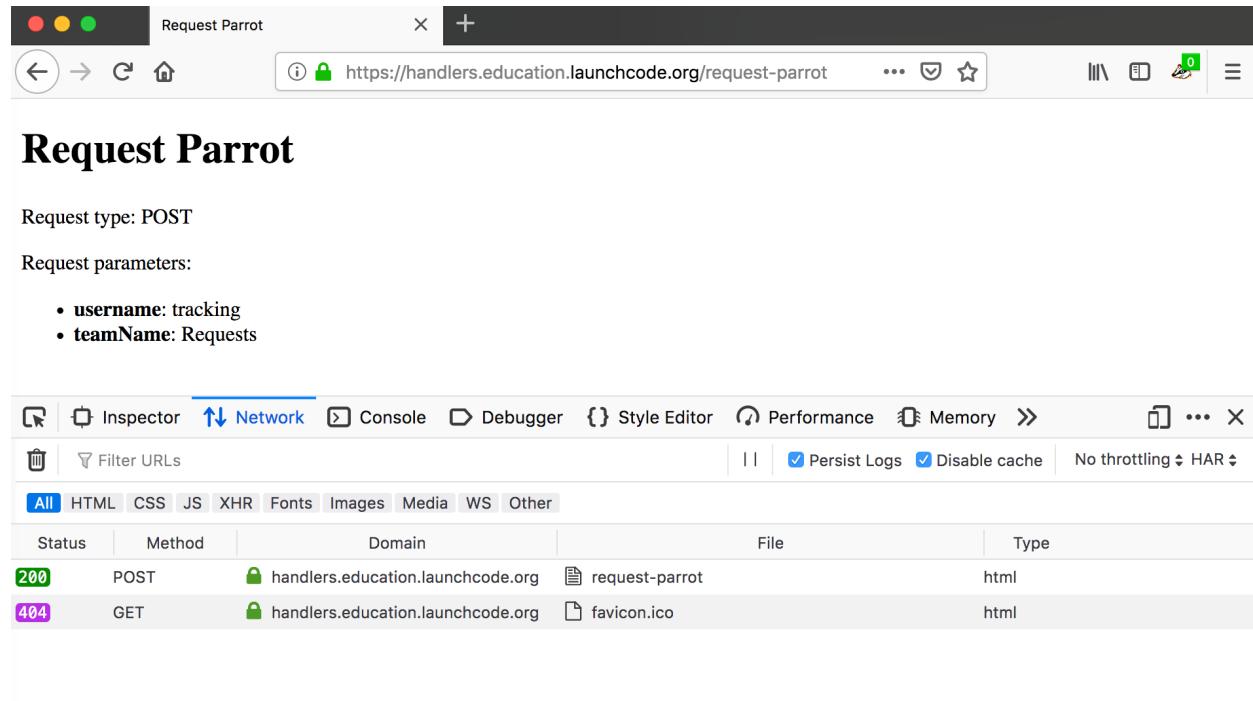


Fig. 2: Firefox browser with request handler loaded and network tab showing requests

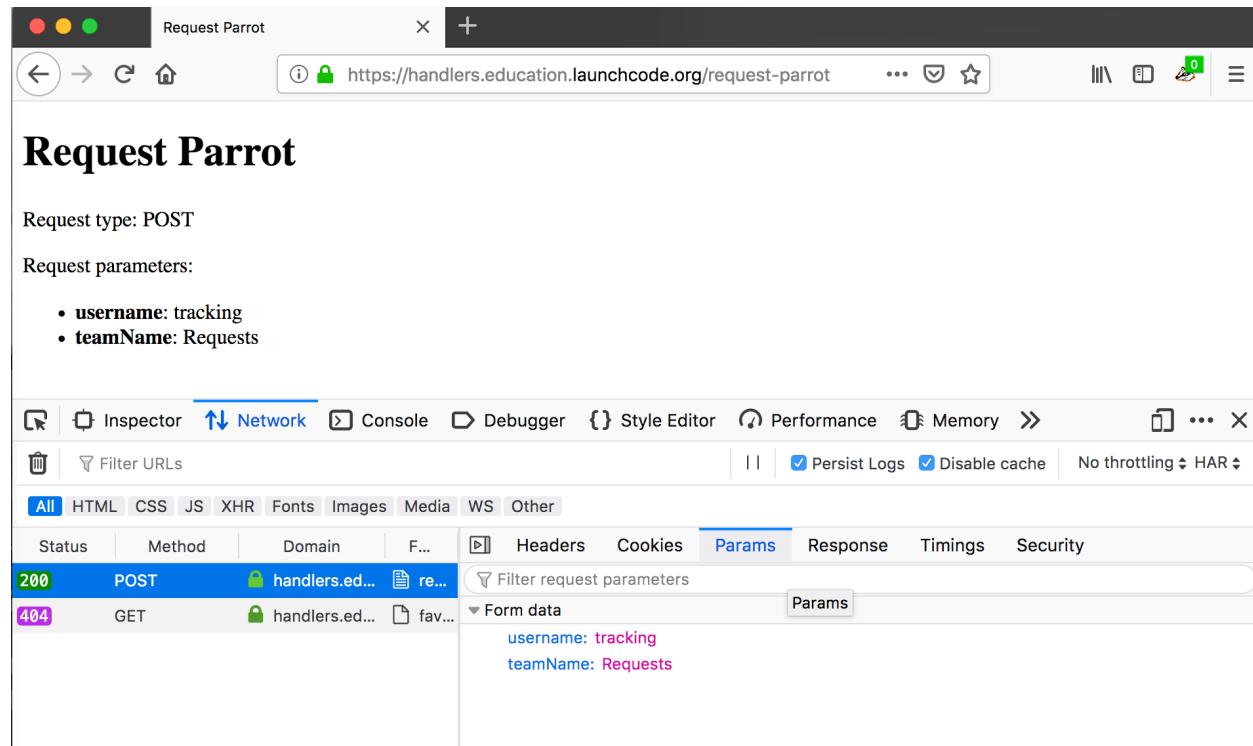


Fig. 3: POST request highlighted with Params tab open showing Form data

Question

What attribute on <form> determines *where* the request is sent?

Question

What do *form handlers* do with form submissions?

25.4 Text Inputs

As you know from interacting with web forms, it's possible to use more than simple text inputs. There are additional input *types*, each with different uses. Many of the elements are <input> tags with a different type value, however some have entirely different tag names. The next few sections contain lists of input types.

To start, here are three types of text inputs. These input types can contain text of any value.

Type	Syntax	Description	Demo
text	<input type="text" name="username"/>	A single line text field.	
textarea	<textarea name="missionDescription"></textarea>	A larger, multi-line text box. Must have open and closing tags.	
password	<input type="password" name="passCode"/>	A text field that obscures the text typed by the user.	

Note

Form inputs will NOT look exactly the same in all browsers. However, the inputs *should* function the same way. Use <https://caniuse.com>, if there is ever a question of browser support for a certain feature.

25.4.1 Example

Example

```
form action "https://handlers.education.launchcode.org/request-parrot" method "post"
    label Code Name input type "text" name "codeName"    label
    label Code Word input type "password" name "codeWord"    label

    
    label Mission Description br
        textarea name "description" rows "5" cols "75"    textarea
    label

    button Send Report button
form
```

Submitted Values

Code Name

Code Word

Mission Description

`Test flight.
Plane maintenance.
Superhero stuff.`

```
codeName=Captain+Danvers  
codeWord=avengers!  
description=Test+flight.+Plane+maintenance.+Superhero+stuff.
```

Notice that the textarea value does NOT include new lines, even though it was typed that way.

[Run it](#)

25.4.2 Check Your Understanding

Question

Which input type should be used if the user is going to enter a large amount of text?

25.5 Specialized Text Inputs

For these text inputs the browser will validate and provide feedback to the user based on rules for the declared type.

Type	Syntax	Description	Demo
date	<code><input type="date" name="flightDate"/></code>	Browser validates the value is a valid date format. Some browsers provide a <i>date picker</i> .	
email	<code><input type="email" name="emailAddress"/></code>	Browser validates the value is a valid email address format.	
number	<code><input type="number" name="fuelTemp"/></code>	Browser validates the value is a valid number format.	

25.5.1 Example

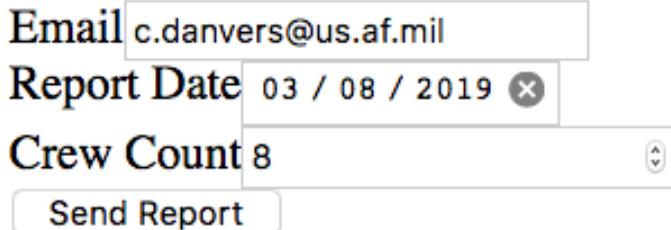
Example

```
form action "https://handlers.education.launchcode.org/request-parrot" method "post"  
label Email input type "email" name "emailAddress" label  
label Report Date input type "date" name "reportDate" label  
label Crew Count input type "number"
```

(continues on next page)

(continued from previous page)

```
name "crewCount" min "1" max "10"    label  
button Send Report button  
form
```



Email c.danvers@us.af.mil

Report Date 03 / 08 / 2019

Crew Count 8

Submitted Values

```
emailAddress=c.danvers@us.af.mil  
reportDate=2019-03-08  
crewCount=8
```

Run it

25.5.2 Check Your Understanding

Question

What happens if you type in 1234 into <input type="email"/>?

25.6 Checkbox Input

A checkbox input represents a box to check. Checkbox inputs can be used by themselves or in groups. Checkbox inputs are best used with <label> tags.

Type	Syntax	Description	Demo
checkbox	<input type="checkbox" name="signUp" />	A small box for marking form option as <i>checked</i> .	

25.6.1 Examples

Example

One checkbox. No value attribute is set, so the default value of on is submitted.

```
label crew input type "checkbox" name "crewReady" label
```

Submitted (if checked)

```
crewReady=on
```

Run it

Example

Multiple checkbox inputs. All with *different* name attributes.

```
div Activities div
label cooking input type "checkbox" name "cooking" label
label running input type "checkbox" name "running" label
label movies input type "checkbox" name "movies" label
```

Submitted (if cooking and movies are checked)

```
cooking=on&movies=on
```

Run it

Example

Multiple checkbox inputs with the SAME name attribute.

```
div Ingredients div
label Onion input type "checkbox" name "ingredient" value "onion" label
label Butter input type "checkbox" name "ingredient" value "butter" label
label Rice input type "checkbox" name "ingredient" value "rice" label
```

Submitted (if butter and rice are checked)

```
ingredient=butter&ingredient=rice
```

Run it

25.6.2 Check Your Understanding

Question

What is the default value submitted for a <checkbox> when checked?

25.7 Radio Input

Radio inputs allow a user to pick one option out of a grouping of options. Radio inputs with the same name are grouped. Only one radio input in a group can be chosen at a time. The `value` attribute of the chosen radio input will be submitted. Radio inputs are best used with `<label>` tags.

Type	Syntax	Description	Demo
radio	<code><input type="radio" name="crewReady" value="yes" /></code>	A small circle that allows selecting <i>one</i> of multiple values. Used in groups of two or more.	

25.7.1 Example

Example

```
form action "https://handlers.education.launchcode.org/request-parrot" method "post"
Flight Rating:
  label Rough input type "radio" name "flightRating" value "rough"   label
  label Few Bumps input type "radio" name "flightRating" value "fewBumps"   label
  label Smooth input type "radio" name "flightRating" value "smooth"   label
  button Send Report  button
form
```

Flight Rating:

Rough

Few Bumps

Smooth

Send Report

Submitted Values

```
flightRating=smooth
```

Run it

25.7.2 Check Your Understanding

Question

Should a group of radio inputs have the same value for the `name` attribute?

25.8 Select Input

Select inputs create a clickable menu that displays options and allows the user to select one. The available options are defined by `<option>` tags inside of the `<select></select>` tag.

`<option>` tags have a `value` attribute which defines the value submitted if that option is selected. The text inside the `<option>Option text</option>` is what is displayed in the select menu.

Type	Syntax	Description	Demo
select	<code><select name="weather"><option value="1">clear</option><option value="2">cloudy</option></select></code>	A menu that allows selection of one option. Requires options to be in <code><option></code> tags.	

25.8.1 Example

Example

```
form action "https://handlers.education.launchcode.org/request-parrot" method "post"
  label Operation Code:
    <!-- includes empty value "Select One" option -->
    select name "operation"
      option value "" * Select One * option
      option value "1" Simulation option
      option value "2" Rocket Test option
      option value "3" Crew Related option
    select
  label

  label Facility:
    select name "facility"
      option value "johnson" Johnson Space Center, TX option
      option value "kennedy" Kennedy Space Center, FL option
      option value "white-sands" White Sands Test Facility, NM option
    select
  label
  button Send Report button
form
```

Default Form Values



Select “Rocket Test” and “White Sands Test Facility, NM”

Submitted Values

```
operation=2
facility=white-sands
```

Operation Code:

Facility:

Run it

25.8.2 Check Your Understanding

Question

For a select input, what determines the value that is submitted during form submission?

25.9 Validation with JavaScript

Validating form inputs *before* submitting the form can make the user experience much smoother. Some input types have built-in browser validation for basic formats such as numbers and email addresses. We can use event handlers to perform more complex validation on form input values.

25.9.1 Form Inputs and the DOM

Before we can validate what the user has typed we need to understand how to use form inputs with the DOM. Remember that the DOM is a JavaScript representation of the HTML document. `<input>` tags can be selected and referenced like any other HTML element.

To read the value of an `input`, we can check the `value` attribute. We can also assign a new value to `input.value` which will update the value shown in the input.

Example

This example will log the value of an input, update the input's value, and then log it again when the button is clicked.

```
1  <!DOCTYPE html>
2  html
3      head
4          title Check input value with DOM  title
5      head
6      body
7          form
8              label Language
9                  input type "text" name "language" id "language" value "JavaScript"
10             label
11            form
12                button id "update" Update Input Value  button
13                script
14                    let
15                        "update"
```

(continues on next page)

(continued from previous page)

```
15 // add event handler for when button clicked
16           "click"  function
17     let                      "language"
18
19     // now update the value in the input
20           " rocks!""
21
22
23   script
24 body
25 html
```

repl.it

Question

What happens when you click the button multiple times?

25.9.2 Steps to Add Validation

1. Add an event handler for the `window load` event
2. Within the window's load handler, add an event handler for the `form submit` event
3. Retrieve input values that need to be validated from the DOM.
4. Within the form's submit handler, check the `input` values using conditional statements
 - a. If the values are valid, allow the form submission
 - b. If the values are NOT valid, inform the user and STOP form submission

Each of these steps involves additional details, which we will now break down.

Example

Let's start this by showing an `alert` box when the form `submit` event is triggered.

```
1 html
2   head
3     title Form Validation  title
4     style
5       label display block
6       body padding 25px
7     style
8   head
9   script
10          "load"  function
11            "form"
12            "submit"  function
13            "submit clicked"
14
15
16   script
```

(continues on next page)

(continued from previous page)

```

17   body
18     form method "POST" action "https://handlers.education.launchcode.org/request-
19   ↪parrot"
20       label Username input type "text" name "username" label
21       label Team Name input type "text" name "team" label
22       button Submit button
23   form
24 body
html

```

25.9.3 Follow Along as We Add Validation

Use [this repl.it](#) and the following instructions to add validation to the above example.

Get Reference to Inputs

To validate what the user has typed, we can get a reference to the `input` elements in the DOM and check the `value` property of each. Let's change the `submit` event handler to display the value of the `username` input in an `alert` box. To do that, we are going to use `document.querySelector("input[name=username]")`, which uses an *attribute selector* to select the `<input>` that has `name="username"`.

```

1 script
2           "load"  function
3     let          "form"
4           "submit"  function
5         let          "input [name=username]"
6           // alert the current value found in the username input
7           "username: "
8
9
10    script

```

Alert the Input Values When Submitted

Now that we know how to get the value of an `input`, we can add *conditional statements*. Let's add code that opens an `alert` box if *either* input value is *empty*.

```

1 script
2           "load"  function
3     let          "form"
4           "submit"  function
5         let          "input [name=username]"
6           let          "input [name=team]"
7           if          ""
8             "All fields are required!"
9
10
11    script

```

We are making progress. Now if you click `Submit` with one or both of the inputs empty, then an alert message appears telling you that both inputs are required. However, the form is still submitted even if the data is invalid.

Prevent Form Submission

We should prevent the form submission from happening until all inputs have valid values. We can use the `event` parameter and `event.preventDefault()` to stop the form submission. `event.preventDefault()` prevents default browser functionality from happening, like form submission when `<button>` tags are clicked inside of a form. Remember that *event handler* functions are passed an `event` argument which represents the event that the handler is responding to.

```
1  script
2          "load"  function
3      let           "form"
4          "submit"  function
5      let           "input [name=username]"
6      let           "input [name=team]"
7      if           ""
8          "All fields are required!"
9      // stop the form submission
10
11
12
13
14  script
```

25.9.4 Check Your Understanding

Question

What event should you listen to if you want to validate a form before it's submitted?

Question

What method on the `event` object can be used to stop a form submission?

25.10 Exercises: Forms

Hello programmer, we need you to make a Rocket Simulation form. Please follow the steps below and good luck!

Code your solution in [this repl.it](#).

1. Create a `<form>` with these attributes.
 - Set `method` to "POST"
 - Set `action` to "http://handlers.education.launchcode.org/request-parrot"
2. Add a `<label>` and `<input>` for Test Name to the `<form>`.
 - `<label>Test Name <input type="text" name="testName"/></label>`.

Display Name	Input Type	Input Name	Possible Values
Test Name	text	testName	No limitations

3. Can you submit the form now? What is missing?
4. Add a `<button>Run Simulation</button>` to the `<form>`.

5. Enter a value into the “testName” input and submit the form.

- Was the value properly submitted to the form handler?

6. Next add these five inputs to the <form>.

- Pay attention to the types and possible options.
- Also add a <label> for each input.

Display Name	Input Type	Input Name	Possible Values
Test Date	date	testDate	Date format mm/dd/yyyy
Rocket Type	select	rocketType	Brant, Lynx, Orion, Terrier
Number of Rocket Boosters	number	boosterCount	A positive number less than 10
Wind Rating	radio	windRating	No Wind: with value 0, Mild: with value 10, Strong: with value 20
Use production grade servers	checkbox	productionServers	on or off

Example

What the form will look like *before* submission.

Rocket Simulation

Test Name

Test Date

Rocket Type

Number of Rocket Boosters

Wind Rating:

No Wind

Mild

Strong

Use production grade servers

Submitted Values

```
testName=Moon+Shot
testDate=2020-07-16
rocketType=Lynx
boosterCount=3
windRating=10
productionServers=on
```

25.10.1 Bonus Mission

Use an event handler and the `submit` event to validate that all inputs have values. Do NOT let the form be submitted if inputs are empty.

25.11 Studio: HTTP and Forms

25.11.1 Introduction

This chapter taught you that forms submit data in HTTP requests. This studio uses form and HTTP concepts to build a *search engine selector*, that is, a search form that allows a user to choose which search engine they would like to use. It will look like this:



Fig. 4: The search engine selector that we will build.

Most search engines work the same way. They have a single text input, and they submit data using a GET request. Additionally, many of the most popular search engines also use the same name for the search parameter, `q`.

Try It!

Use 2-3 different search engines to search for the same term. On the results page, look at the URL. Did the search happen via GET or POST? If a GET request was made, what is the name of the parameter containing your search term?

Note: You may have to copy/paste the URL into a text editor to find the search parameter. Some engines add other parameters to the URL, causing it to extend past the end of the browser's address bar.

Note

We remarked previously that most forms use POST because they cause data to be changed on the server. A web search only *retrieves* data. It does not change data. Therefore it's safe to use a GET request for searches.

The fact that most search engines use the name `q` for their search boxes will allow us to easily create a form that is capable of sending a search request to several search engines.

The form will send a request with query parameter `q` to the selected engine. Since this request looks essentially the same as requests coming from the search engine's own form (for example, at google.com) it will give us back the results the same as if we had searched via those sites.

25.11.2 Getting Started

- Create a new directory, `forms-studio`
- `cd` into this new directory
- Create a file to use for the studio: `touch index.html`
- Copy/paste the [starter code](#) for the studio into `index.html`

25.11.3 Create Form Inputs

Let's build out the form. We will need some data for the search engines we want to work with.

Table 1: Search Engine Options

Label	Value	Search URL
Google	google	https://www.google.com/search
DuckDuckGo	duckDuckGo	https://duckduckgo.com/
Bing	bing	https://www.bing.com/search
Ask	ask	https://www.ask.com/web

- Create a text input within the form and set its name attribute to the value "q".
- Create a radio group with one radio button for each search engine. Recall that radio buttons with the same name are grouped, so use the same value for this attribute, "engine", on each radio button.
- Create a label element for each radio button.
- Finally, add a submit button to the form and set its value to "Go!".

Question

How is the value attribute of a submit button used?

25.11.4 Submit Event Handler

Pop quiz:

Question

What happens if you try to submit the form at this point? Why?

Question

Which HTTP method will be used when submitting the form?

We now have a form with inputs that has nowhere to send its data. The `action` attribute determines where a form submits data, but we can't set the `action` attribute on the form in our HTML. Our form needs to submit data to a different site based on the selected search engine.

To make this happen, we need to set the value of the form's `action` *after* the user hits the submit button, but *before* the form request is sent. Forms trigger a `submit` event at precisely this moment. Therefore, we can create an event handler to solve this problem. Our handler will:

1. Retrieve the selected value from the radio group.
2. Use this value to determine the `action` URL, based on the selected search engine.
3. Set the `action` attribute of the form.

Create and Register the Handler

Within the `<script>` element near the top of the file, create a function named `setSearchEngine`. We will code this function later, so for now just add a `console.log` statement so we can see when it runs.

Near the bottom of the `<script>` element is the stub:

```
1           'load'  function
2   // TODO: register the handler
3
```

Replace the TODO with code to add `setSearchEngine` as a handler to the form's submit event. You will first need to get the form element using one of the DOM methods.

Note

The event handler can be added only after the form has been built, so we do so by adding a `load` event handler to the `window`. This ensures that the event is registered *after* the page has loaded.

Before moving on, make sure the code you just wrote works. Submit the form and look for a message in the console to verify that `setSearchEngine` ran.

Set the action

Our event handler now runs when the form is submitted, but it doesn't do anything. We would like it to set the `action` on the form based on the user's choice of search engine.

Add code to `setSearchEngine` to get the selected radio button element, using `document.querySelector`. The selector you'll need is a little complicated, so we'll give it to you here:

```
input [name=engine]:checked
```

This compound CSS selector combines an *attribute* selector with a *pseudo selector*. The attribute selector `input [name=engine]` matches all `input` elements with the attribute `name` equal to "engine". The pseudo selector `:checked` specifies that we only want the selected element from that group of matches. Combined, the selector gives us the selected element in the radio group.

Once you have the selected radio button, get its value using `.value`. The value tells us which search engine the user has chosen.

At this stage, we could use a large `if/else if/else` statement to determine the URL for the selected search engine.

```
let actionURL;

if (engine === "google") {
  actionURL = "https://www.google.com/";
} else if (engine === "bing") {
  actionURL = "https://duckduckgo.com/";
}

// ... and so on ...
```

This is ugly and inefficient. A better approach is to create an object to store the engine values and URLs as key/value pairs. For a single engine, the object would look like:

```
1 let
2   "google"  "https://www.google.com/"
3
```

Add this to your code, and fill it out to include the other three engines.

Now, you can get the action URL using `action`, bracket notation, and the value of the selected radio button. Once you have the action URL, find the form element and set its action using `setAttribute`.

If everything went well, your search engine selector page should now work! If not, that's okay. Switch to debugging mode and figure out what needs fixing.

25.11.5 Bonus Missions

1. Add validation to your submit handler to make sure that the user has both selected a search engine and entered a (non-empty) search term.
2. Add some CSS rules to your page to make it look nice.

FETCH & JSON

26.1 Introduction

One of the main benefits of programming is we don't work in isolation. We can import modules that contain code that we can use to help us write our own programs. We use online documentation like MDN and W3Schools to help us learn how to utilize aspects of a programming language. We use online forums like Stack Overflow and Google to find answers to specific problems. We even use other people like our classmates, TAs, and instructors to help figure out how to solve problems.

We can also use other people's data in our applications. There are multiple ways of using other people's data, or external data, in our applications. In this chapter, we will focus on using `fetch()` and `JSON` to request and use data.

26.1.1 API

When using a website, we mainly work with **GUIs (Graphical User Interface)** which contain buttons, forms, text boxes, etc. However, our program does not know how to use a GUI. Programs use **APIs (Application Programming Interface)** need an to communicate with other programs.

Consider the software you use on a daily basis, like Microsoft Word, Google Chrome, or a music streaming device like Spotify. When you open the software, a window pops up on your screen and is filled with text, buttons, search bars, scroll bars, etc. Usually, with a little trial and error, you can learn how to use the interface easily. This interface we use is called a Graphical User Interface, or GUI for short.

How we interact with computers is with various interfaces, either GUI, or CLI. However, an application does not know how to use a GUI, or a CLI, and needs it's own interface to communicate with another application. So a new interface was defined so that one application could communicate with another application. This interface was called an Application Programming Interface, or API for short.

An API is how one application communicates with another application. We will be making a request to an API in order to retrieve information we need for our application.

26.2 Data Formats and JSON

In order for our application to make a request to an API, the data will need to be formatted in a way both our application and the API can understand.

The API may have been built with another programming language. And it may not use the same variables, objects, and data structures as JavaScript. To solve the issue of APIs being built in different programming languages, data formats are used.

A **data format** is a set of rules that govern how data is written, organized, and labeled. Data formats make working with data consistent and reliable.

26.2.1 JSON

There are quite a few different data formats, but we will only focus on one throughout this class: **JavaScript Object Notation**, also known as **JSON**. JSON is one of the leading data formats used, especially on the web.

JSON is based on JavaScript object syntax, but has some differences.

Let's consider an API that serves information about the books in a library. In this example, we searched for "An Astronaut's guide to life on Earth".

```
1   "title"  "An Astronaut's guide to life on Earth"
2   "author"  "Chris Hadfield"
3   "ISBN"    9780316253017
4   "year_published"  2013
5   "subject"  "Hadfield, Chris"  "Astronauts"  "Biography"
6   "available"  true
7
8
```

The API returned a match for our search. The search provides us with information that may be useful to the user in the form of the title, author, ISBN, the year the book was published, the subjects of the book, and if the book is currently available for checkout.

26.2.2 JSON Rules

JSON is a collection of key-value pairs. In the example above, "title" is a key and its value is "An Astronaut's guide to life on Earth".

The key-value pairs describe the data that is being transferred.

A JSON key **MUST** be a string, but the value may be a number, string, boolean, array, object, or null.

In the example above, the JSON describes one object, a book! All of the keys are strings, and the values are: string, string, number, array, and boolean respectively.

JSON can also be used to describe a collection of objects at the same time. Consider we search for the word "Astronaut".

```
1   "hits"  3
2   "book"
3
4
5       "title"  "An Astronaut's guide to life on Earth"
6       "author"  "Chris Hadfield"
7       "ISBN"    9780316253017
8       "year_published"  2013
9       "subject"  "Hadfield, Chris"  "Astronauts"  "Biography"
10      "available"  true
11
12
13      "title"  "Astronaut"
14      "author"  "Lucy M. George"
15      "ISBN"    9781609929411
16      "year_published"  2016
17      "subject"  "Astronauts"  "Juvenile Fiction"  "Space stations"
18      "available"  false
19
20
```

(continues on next page)

(continued from previous page)

```

21      "title"  "Astronaut Ellen Ochoa"
22      "author"  "Heather E. Schwartz"
23      "ISBN"    9781512434491
24      "year_published"  2018
25      "subject"  "Ochoa Ellen"  "Women astronauts"  "Astronauts"  "Biography"
26      ↳ "Women scientists"  "Hispanic American women"
27      "available"  true
28
29

```

This time, our search term “Astronaut” returned multiple books, and so a collection of book objects was returned in JSON format.

Each book object can be found in the array with the key “book”. Each book contains the keys “title”, “author”, “ISBN”, “year_published”, “subject”, and “available”.

When we make a request to an API, the API formats the data we requested into JSON and then responds to our request with the JSON representation of our request.

26.2.3 JSON & JavaScript Object Differences

JSON is rooted in JavaScript objects syntax. However, there are some key differences between the two.

JSON keys MUST be in double quotes. Double quotes should not be used when declaring properties for a JavaScript object.

JSON:

```

1
2      "title"  "The Cat in the Hat"
3      "author"  "Dr. Seuss"
4

```

JavaScript object:

```

1 let
2     "The Cat in the Hat"
3     "Dr. Seuss"
4

```

To represent a string in JSON, you MUST use double quotes. In JavaScript, you can use double quotes or single quotes.

JSON:

```

1
2      "title"  "The Last Astronaut"
3      "author"  "David Wellington"
4

```

JavaScript object:

```

1 let
2     'The Last Astronaut'
3     'David Wellington'
4

```

Note

JSON is based on JavaScript objects, but there are key differences. JSON syntax is a little more strict than JavaScript object syntax.

26.2.4 Check Your Understanding

Question

What does API stand for?

Question

Why might you connect to an API?

Question

What is JSON?

Question

What purpose does JSON serve?

26.3 Fetching Data

Now that we know what an API is, let's use one to update a web page. Let's use a weather API to add weather data to a webpage. The URL for this special LaunchCode weather API is <https://api.education.launchcode.org/weather>.

Example JSON returned from our weather API.

```
"temp" 67
"windSpeed" 5
"tempMin" 50
"tempMax" 71
"status" "Sunny"
"chanceOfPrecipitation" 20
"zipcode" 63108
```

We can see that this API returns useful information like `temp` and `windSpeed`. Our goal is to add that data to a Launch Status web page. Note, this API is for instruction purposes and does not contain real time data.

Example

Launch Status web page, which we will add weather data to.

```
1 <!DOCTYPE html>
2   html
3     head
4       title Launch Status title
5     head
6   body
7     h1 Launch Status h1
8     h3 Weather Conditions h3
9     div id "weather-conditions"
10      <!-- TODO: dynamically add html about weather using data from API --&gt;
11      div
12    body
13  html</pre>
```

Warning

Before going through the `fetch` examples, please know that `fetch` does NOT work in Internet Explorer. List of alternative browsers

26.3.1 `fetch` Function

To request the weather data, we will use the `fetch` function. `fetch` is a global function that requests, or fetches, resources such as data from an API.

Take note of two necessary aspects of the `fetch` function:

1. The URL of where the data is located.
 - For this example it will be '`https://api.education.launchcode.org/weather`'
2. A response handler function to utilize the data that is being fetched.
 - For this example it will be `function(response) { ... };`

Example

Notice a string URL is passed to `fetch`. Also notice the anonymous function that has a `response` parameter, that is the request handler function. The `then` function will be explained soon.

```
'https://api.education.launchcode.org/weather'      function
```

In this example, we are requesting data from `https://api.education.launchcode.org/weather` and our response handler (the function) simply logs the response to the console.

26.3.2 `fetch` Example

Now let's add `fetch` in the Launch Status web page.

Example

A <script> tag has been added that includes:

1. A *load* event handler on line 5
2. A `fetch` and response handler function on line 6
3. A `console.log(response);` on line 7 that prints out the response object

```
1  html
2    head
3      title Launch Status  title
4      script
5        "load"  function
6          "weather.json"      function
7
8
9
10     script
11   head
12   body
13     h1 Launch Status  h1
14     h3 Weather Conditions  h3
15     div id "weather-conditions"
16       <!-- TODO: dynamically add html about weather using data from API -->
17       div
18   body
19 html
```

repl.it

Let's break down how `fetch` works. A URL is passed to `fetch` as a parameter. That causes an HTTP GET request to be sent from the browser to the API. Remember that HTTP is a request then response protocol. The response handler function, as the name implies, handles the response sent back from the API. Using the data in the response, the web page can be updated using DOM methods.

Note

In this section, `fetch` is used to make GET requests. `fetch` can also be used to make other types of HTTP requests such as POST and PUT.

View the GET Request

We can see evidence of the GET request by following these steps:

1. Open the [Launch Status web page](#) in its own tab.
2. Open developer tools.
3. Open the *Network* tab in developer tools.

In the above image, you can see the web page has been rendered on the left. In the developer tools, the GET request to the Weather API has been highlighted along with the response from that request. The response shows the JSON data that was received. In the console output, you can see the `Response` object has been logged. We will use that object next.

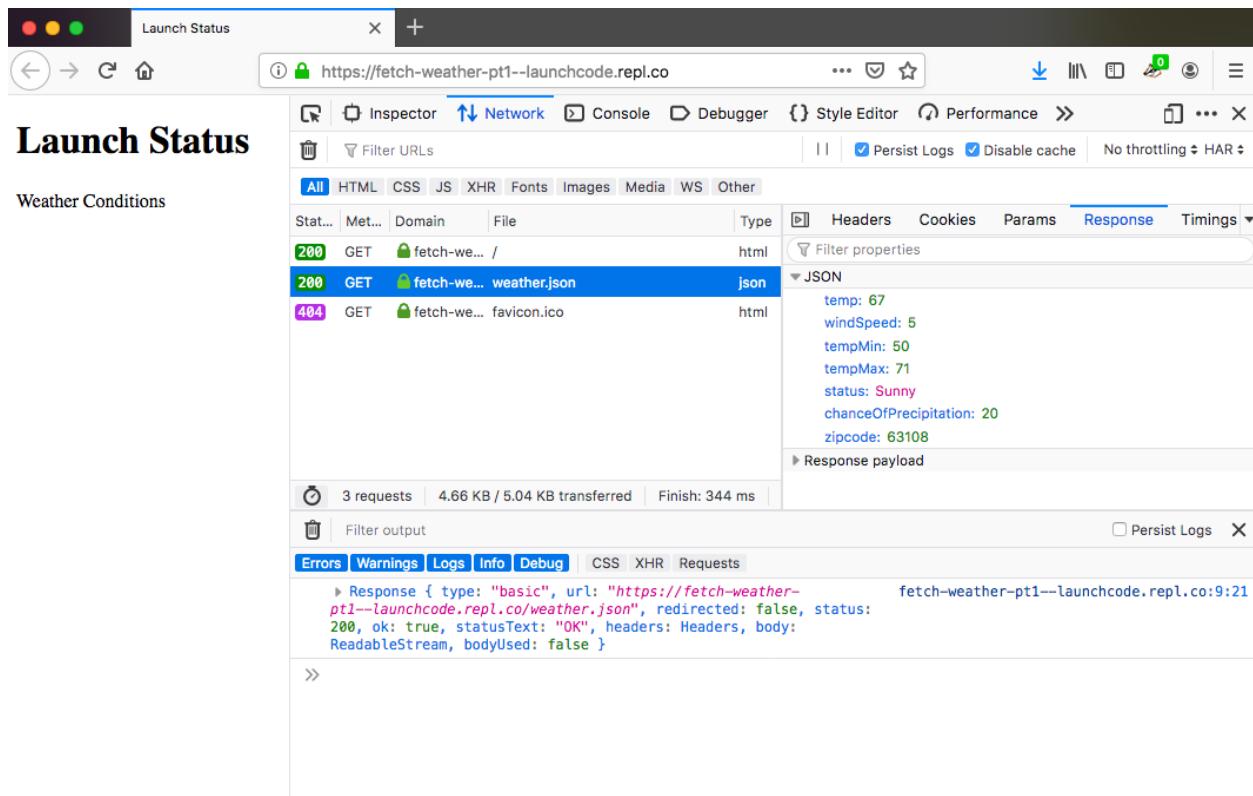


Fig. 1: The GET request to the Weather API highlighted in developer tools.

Response Object

The response to the GET request is contained in a `Response` object that is an instance of the `Response` class. The `Response` class represents an HTTP response and has methods that allow access to the status and data.

Example

On line 8, the `json()` method is used to gain access to the JSON data contained in the response.

Line 9 logs the JSON to the console. We'll discuss `.then()` later.

```

1  html
2      head
3          title Launch Status  title
4          script
5              "load"  function
6                  "weather.json"      function
7                      // Access the JSON in the response
8                          function
9
10
11
12
13          script
14      head
15      body
16          h1 Launch Status  h1

```

(continues on next page)

(continued from previous page)

```
17  h3 Weather Conditions  h3
18  div id "weather-conditions"
19  <!-- TODO: dynamically add html about weather using data from API -->
20  div
21  body
22  html
```

repl.it

Console Output

```
Object { temp: 67, windSpeed: 5, tempMin: 50, tempMax: 71, status: "Sunny", ↵
↪chanceOfPrecipitation: 20, zipcode: 63108 }
```

Use the DOM and JSON Data to Update the Page

Now that we have JSON weather data, we can add HTML elements to the page to show the data.

Example

On line 10, `innerHTML` of the `div` variable is set to be HTML built using JSON weather data.

```
1  html
2   head
3     title Launch Status  title
4     script
5       "load"  function
6       "weather.json"      function
7         function
8         const          'weather-conditions'
9         // Add HTML that includes the JSON data
10        `
11
12        <ul>
13          <li>Temp ${}      </li>
14          <li>Wind Speed ${}      </li>
15          <li>Status ${}      </li>
16          <li>Chance of Precip ${}      </li>
17        </ul>
18
19
20
21        script
22      head
23    body
24      h1 Launch Status  h1
25      h3 Weather Conditions  h3
26      div id "weather-conditions"
27        <!-- Weather data is added here dynamically. -->
28        div
29      body
30      html
```

repl.it

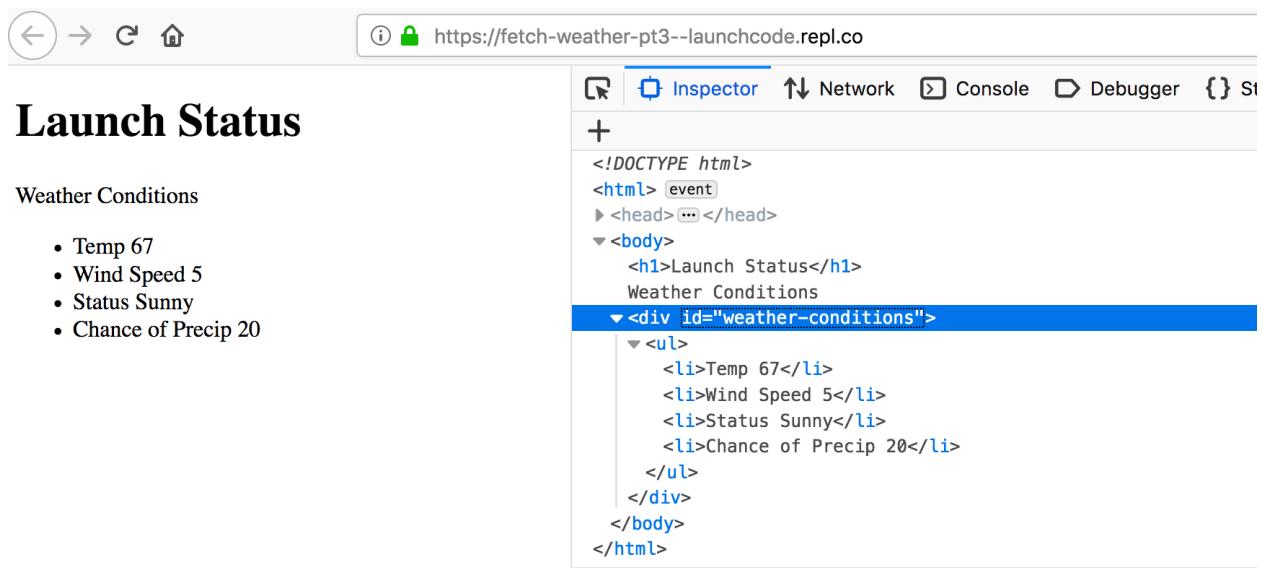


Fig. 2: Weather data added to web page.

Note

`fetch` was chosen as the tool to request data because it's supported in modern browsers by default and is simple to use. When viewing resources other than this book, you will see various other ways to request data in a web page with JavaScript. Other ways include, but are not limited to, `jQuery.get`, `jQuery.ajax`, and `XMLHttpRequest`.

26.3.3 Check Your Understanding

Question

What is the correct syntax for `fetch`?

- `fetch('GET', 'https://api.education.launchcode.org/weather').then(...);`
- `fetch('https://api.education.launchcode.org/weather').doStuff(...)`
- `fetch('https://api.education.launchcode.org/weather').then(...);`

26.4 Asynchronous and Promises

In order to fully explain how the `fetch` function works, we need to define and talk about the terms **asynchronous** and **synchronous**.

Asynchronous: Not simultaneous or concurrent in time.

Synchronous: Simultaneous or concurrent in time.

When fetching data in JavaScript, the HTTP requests are asynchronous. In brief, that means when an HTTP request is sent, we don't know exactly when a response will be received by the browser. Remember that HTTP requests are sent

to an address, then a response is sent. That process takes a variable amount of time depending on network speed, the address location, and response size.

Note

These requests are also called **AJAX requests** (Asynchronous JavaScript and XML). The XML part of AJAX refers to a data format that was popular before JSON.

26.4.1 Response Handlers

Browsers can't stop everything and wait for a response to an HTTP request. Browsers have to render HTML, interact with the user, and run JavaScript. To keep these processes running seamlessly, without any noticeable pauses, the browser relies on events.

This is where `.then()` and the response handler function come in. The browser provides us with a way to handle the response whenever it is received.

26.4.2 Promises and the `then` Function

Let's look again at a simple `fetch` example. Notice on line 1 that `then` is called on the value returned from `fetch`.

```
1  'https://api.education.launchcode.org/weather'      function
2
3
```

To make it clearer, let's capture the value returned by `fetch` in a variable named `promise`.

```
1 const          'https://api.education.launchcode.org/weather'
2   function
3
4
```

`fetch` returns an instance of the **Promise** class. The **Promise** class represents a future event that will eventually occur. In the above example, `promise` represents the eventual response from the HTTP request to `https://api.education.launchcode.org/weather`.

A promise can be fulfilled or rejected. When a promise is fulfilled, data is passed to the response handler function. The `then` method of a promise defines what will happen when the promise is fulfilled. When a promise is rejected, an error reason is provided.

The above example can be translated to these steps

1. Make an HTTP request to `https://api.education.launchcode.org/weather`
2. When the response is received, THEN run the response handler function (passing in response data)
3. In the response handler function, console log the response object

26.4.3 More Promises

Promises are used for more than HTTP requests. A promise can represent any future event. The `response` object has a `json()` function that will return the JSON data in the response. The `json()` function returns a promise that represents the future result of turning the response data into JSON.

The below example involves two promises. One promise on line 1 that represents the fetch request and a second on line 3 that represents the response data being turned into JSON.

Finally on line 5, the JSON data can be logged.

```
1 const          'https://api.education.launchcode.org/weather'  
2   const         function  
3     const        function  
4       "temp"  
5  
6  
7
```

Tip

Promises can be a hard concept to understand. Focus on the examples and the theory will make sense in time.

26.4.4 Check Your Understanding

Question

True or False, we know exactly when an asynchronous request will return?

Question

True or False, a Promise can represent any future event?

Question

True or False, `then` is a method of the `Promise` class that allows us to run code after an event is completed?

26.5 Exercises

26.5.1 JSON

1. Which of the following three code snippets is correct JSON syntax? Why are the other two options incorrect?

```
"dog"  
"Bernie"  
3
```

```
"type"  "dog"  
"name"  "Bernie"  
"age"   3
```

```
"type"  "dog"
"name"  "Bernie"
"age"   3
```

2. Which of the following three code snippets is correct JSON? Why are the other two options incorrect?

```
"animals"

  "type"  "dog"
  "name"  "Bernie"
  "age"   3

  "type"  "cat"
  "name"  "Draco"
  "age"   2
```

```
"type"  "dog"
"name"  "Bernie"
"age"   3
```

```
"type"  "cat"
"name"  "Draco"
"age"   2
```

```
"type"  "dog"
"name"  "Bernie"
"age"   3
```

```
"type"  "cat"
"name"  "Draco"
"age"   2
```

26.5.2 Fetch

Todo: add exercises for fetch

26.6 Studio: Fetch & JSON

Your task is to build a website that shows astronauts fetched from an API. Requirements are below.

Repl.it with starter code

26.6.1 Requirements

1. Add code that runs on the `window load` event.
 - This is done because we can't interact with the HTML elements until the page has loaded.
2. Make a GET request using `fetch` to the astronauts API <https://api.education.launchcode.org/astronauts>
 - Do this part inside the `load` event handler.
3. Add each astronaut returned to the web page.
 - Use this HTML template shown below.
 - Be sure to use the exact HTML including the CSS classes. (starter code contains CSS definitions)

Example JSON

Notice that it's an array of objects, due to the outer [and]. That means you will have to use a loop to access each object inside the JSON array.

```

1
2
3     "id"  1
4     "active"  false
5     "firstName"  "Mae"
6     "lastName"  "Jemison"
7     "skills"
8         "Physician"  "Chemical Engineer"
9
10    "hoursInSpace"  190
11    "picture"  "mae-jemison.jpg"
12
13
14    "id"  2
15    "active"  false
16    "firstName"  "Frederick"
17    "lastName"  "Gregory"
18    "skills"
19        "Information Systems"  "Shuttle Pilot"  "Fighter Pilot"  "Helicopter Pilot
20    ↵  "Colonel USAF"
21
22    "hoursInSpace"  455
23    "picture"  "frederick-gregory.jpg"
24
25
26    "id"  3
27    "active"  false
28    "firstName"  "Ellen"
29    "lastName"  "Ochoa"
     "skills"
```

(continues on next page)

(continued from previous page)

```
30     "Physics"  "Electrical Engineer"
31
32     "hoursInSpace"  979
33     "picture"  "ellen-ochoa.jpg"
34
35
36     "id"  4
37     "active"  false
38     "firstName"  "Guion"
39     "lastName"  "Bluford"
40     "skills"
41         "Aerospace Engineer"  "Philosophy"  "Physics"  "Colonel USAF"
42         "Fighter Pilot"
43
44     "hoursInSpace"  686
45     "picture"  "guion-bluford.jpg"
46
47
48     "id"  5
49     "active"  false
50     "firstName"  "Sally"
51     "lastName"  "Ride"
52     "skills"
53         "Physicist"  "Astrophysics"
54
55     "hoursInSpace"  343
56     "picture"  "sally-ride.jpg"
57
58
59     "id"  6
60     "active"  true
61     "firstName"  "Kjell"
62     "lastName"  "Lindgren"
63     "skills"
64         "Physician"  "Surgeon"  "Emergency Medicine"
65
66     "hoursInSpace"  15
67     "picture"  "kjell-lindgren.jpg"
68
69
70     "id"  7
71     "active"  true
72     "firstName"  "Jeanette"
73     "lastName"  "Epps"
74     "skills"
75         "Physicist"  "Philosophy"  "Aerospace Engineer"
76
77     "hoursInSpace"  0
78     "picture"  "jeanette-epps.jpg"
79
80
```

HTML Template

Create HTML in this exact format for each astronaut, but include data about that specific astronaut. For example the below HTML is what should be created for astronaut Mae Jemison. All HTML created should be added to the <div>

```
id="container">> tag.
```

Do NOT copy and paste this into your HTML file. Use this as a template to build HTML dynamically for each astronaut returned from the API.

```
1  div class "astronaut"
2    div class "bio"
3      h3 Mae Jemison  h3
4      ul
5        li Hours in space: 190  li
6        li Active: false  li
7        li Skills: Physician, Chemical Engineer  li
8      ul
9    div
10   img class "avatar" src "images/mae-jemison.jpg"
11 
```

Expected Results

What the web page should look like after your code loads the data and builds the HTML.

26.6.2 Bonus Missions

- Display the astronauts sorted from most to least time in space.
- make the “Active: true” text green, for astronauts that are still active (active is true).
- Add a count of astronauts to the page.

Astronauts

Mae Jemison

- Hours in space: 190
- Active: false
- Skills: Physician, Chemical Engineer



Frederick Gregory

- Hours in space: 455
- Active: false
- Skills: Information Systems, Shuttle Pilot, Fighter Pilot, Helicopter Pilot, Colonel USAF



Ellen Ochoa



Fig. 3: Example of what resulting page should look like.

CHAPTER
TWENTYSEVEN

TYPESCRIPT

27.1 Why TypeScript?

The final chapters of the first unit cover a framework called Angular. Angular runs on TypeScript for a couple of reasons.

First, TypeScript is a superset of JavaScript. This means that you can write JavaScript code and use it in TypeScript files.

Second, TypeScript is a **statically typed language**. A statically typed language is a language where the type of a variable is given at the time the program is compiled. This is often achieved by adding the type of the variable to the variable declaration. However, this is not what we have been doing so far. JavaScript is a **dynamically typed language**. In a dynamically typed language, the type of the variable is determined at runtime and is based on the value inside the variable, not the variable declaration.

Statically typed languages are considered by many to be more stable and less prone to production errors, because the errors will occur in development.

27.2 Declaring and Using Variables

Since TypeScript is statically typed, the type of value is added to the variable declaration. However, we will still use our `let` and `const` keywords where appropriate.

The general format of a variable declaration is:

```
let
```

27.2.1 number

When declaring a variable and using the `number` type, we add `number` to the variable declaration, like so:

```
let      10
```

27.2.2 string

When declaring a `string`, we want to use the `string` keyword.

```
let      "10"
```

27.2.3 boolean

The `boolean` keyword should be used when declaring a variable of the `boolean` type.

```
1 let          boolean   true
```

27.2.4 Examples

Let's use some familiar variable declarations to compare between what we know how to do in JavaScript and what we are now learning about TypeScript.

```
1 // In JavaScript, we have:  
2  
3 let          "Determination"  
4 let          17500  
5 let          225000000  
6 let          384400  
7 let          0.621  
8  
9 // The same declarations in TypeScript would be:  
10  
11 let          "Determination"  
12 let          17500  
13 let          225000000  
14 let          384400  
15 let          0.621
```

27.2.5 Check Your Understanding

Question

The correct declaration of the variable, `astronautName`, that holds the value, "Sally Ride", is:

- a. `let astronautName = "Sally Ride";`
 - b. `let astronautName = string: "Sally Ride";`
 - c. `let astronautName: string = "Sally Ride";`
 - d. `string astronautName = "Sally Ride";`
-

27.3 Arrays in TypeScript

Arrays in TypeScript must contain values of the same type. When declaring an array, the type needs to be declared.

```
1 let          10 9 8
```

What if the array needs to hold values of different types?

Now, we need a **tuple**. A tuple is a special structure in TypeScript that can hold as many values as needed of different types.

```
1 let
2
3     10 "9" 8
```

27.3.1 Examples

In JavaScript, we would declare an array holding items in our cargo hold like so:

```
let 'oxygen tanks' 'space suits' 'parrot' 'instruction manual' 'meal_
↪packs' 'slinky' 'security blanket'
```

In TypeScript, we would declare the same `cargoHold` array a little differently:

```
let 'oxygen tanks' 'space suits' 'parrot' 'instruction_
↪manual' 'meal packs' 'slinky' 'security blanket'
```

What about declaring arrays for elements on the Periodic Table? In JavaScript, that is a relatively simple task:

```
1 let 'hydrogen' 'H' 1.008
2 let 'helium' 'He' 4.003
3 let 'iron' 'Fe' 55.85
```

In TypeScript, however, an array can only hold values of one type, so we need to use a tuple.

```
1 let
2     'hydrogen' 'H' 1.008
3
4 let
5     'helium' 'He' 4.003
6
7 let
8     'iron' 'Fe' 55.85
```

27.3.2 Check Your Understanding

Question

Which of the following statements is FALSE about a tuple in TypeScript?

- a. Tuples can hold as many elements as needed.
 - b. Tuples hold values of one type.
 - c. When declaring a tuple, programmers include the types of the values in a tuple.
-

27.4 Classes and Interfaces in TypeScript

27.4.1 Classes

Classes in TypeScript look something like this:

```
1 class
2
3     this          " "
4
5
6     return "Hello, "    this
7
8
9
10 let           "Bob" "Smith"
```

You may remember the `this` and `new` keywords from working with classes in JavaScript. Earlier in the chapter, we also noted that when declaring variables in TypeScript, we have to specify the type of value. The same applies to function parameters, as you can see in the constructor.

When *using inheritance*, classes in TypeScript can also use the `extends` keyword to denote child and parent classes, as shown here:

```
1 class
2
3     this
4
5
6
7 class      extends
8     boolean   true
9
10
11
12 let       new      "loud"
13
14
15
```

27.4.2 Interfaces

When we start working with Angular, you may see the `interface` keyword quite a bit. Like classes, interfaces define properties and methods that a type will have. The difference is that interfaces do NOT include initialization of properties or implementations of methods.

We may create an interface for a data type that contains all of the information we need about an astronaut and then use that information in a function.

```
1 interface
2
3
4
5 function
6     return
7
8
9 let           "Bob"
```

Interfaces define the contract that other classes or objects must comply with if implementing that interface. Multiple

classes can implement one interface, and that flexibility allows different classes to share one type. This can be helpful when a function parameter needs to make use of certain behaviors.

```
1  interface
2
3
4
5  class      implements
6
7    this
8
9
```

Example

```
1  interface
2
3
4
5  class      implements
6
7
8
9    this      'rooooaaaarrrr'
10
11
12
13 class      implements
14
15
16
17   this      'ROOOOAAAAARRRRR'
18
19
20
21 function
22   `Panthera says ${ }`
23
24
25 let      new
26 let      new
27
28
29
```

In this example, the `Panthera` interface defines the `roar` property. `Tiger` and `Lion` implement the `Panthera` interface, which means `Tiger` and `Lion` must have a `roar` property.

The function `pantheraSounds` has one parameter of type `Panthera`. The variables `tiger` and `lion` can be passed into `pantheraSounds` because they are instances of classes that implement the `Panthera` type.

Optional Parameters

`null` and `undefined` are primitive data types in TypeScript, however, they are treated differently by TypeScript. If you are planning on using `null` to define a property of an interface that is not known yet, use the TypeScript optional

parameter, ?.

Let's take a look at how that would look in TypeScript.

In JavaScript, we might have an object that looks like so:

```
1 let
2     "Reticulated Giraffe"
3     "Alicia"
4     null
5     10
6     "leaves"
7
```

If we wanted to declare the same object as an interface in TypeScript, we would have to use the optional parameter for the weight property.

```
1 interface
2
3
4
5
6
7
```

27.4.3 export

In TypeScript, you can use the `export` keyword to make classes and interfaces available for import in other files. This will look familiar to you as you saw something similar with *modules*.

Using the `export` keyword looks something like this:

```
1 export class
2     // properties and methods
3
```

27.4.4 import

In TypeScript, you can use the `import` keyword to use classes and interfaces declared in other files available for use in the file you are working on. This is a similar idea to *importing modules*, however, the syntax is different in TypeScript:

```
1 import             'relativefilepath'
2
3 let               new
```

27.4.5 Check Your Understanding

Question

What is the difference between a class and an interface?

27.5 Exercises: TypeScript

27.5.1 Part 1 - Declare Variables With Type

1. Fork the [part 1 repl.it](#).
2. Declare and assign a variable for each in the below table.

Variable Name	Type	Value
spacecraftName	string	'Determination'
speedMph	number	17500
kilometersToMars	number	225000000
kilometersToTheMoon	number	384400
milesPerKilometer	number	0.621

27.5.2 Part 2 - Print Days to Mars

In the *same* repl.it you completed Part 1, do the following.

1. Declare and assign these variables.
 - Remember: variable declarations in TypeScript include the type!
 - `milesToMars` is a number with the value of `kilometersToMars * milesPerKilometer`.
 - `hoursToMars` is a number with the value of `milesToMars / speedMph`.
 - `daysToMars` is a number with the value of `hoursToMars / 24`.
2. Write a `console.log` statement that prints out the days to Mars.
 - Use template literal syntax and the variables `${spacecraftName}` and `${daysToMars}`.

Expected Output

```
Space Shuttle would take 332.67857142857144 days to get to Mars.
```

27.5.3 Part 3 - Create a Function

1. Fork the [part 3 repl.it](#).
2. Define a function that calculates the days it would take to travel.
 - Function name `getDaysToLocation`
 - Parameters
 - `kilometersAway` must be a number.
 - Returns the number of days to Mars.
 - Use the same calculations as in Part 2.1.
 - Inside the function, make the variables name generic. Use `milesAway` and `hours` instead of `milesToMars` and `hoursToMars`.
 - The function should declare that it returns a number.
3. Print out the days to Mars.

- Use template literals, `${getDaysToLocation(kilometersToMars)}` and `${spacecraftName}` .

4. Print out the days to the Moon.

- Use template literals, `${getDaysToLocation(kilometersToTheMoon)}` and `${spacecraftName}` .

Expected Output

```
Space Shuttle would take 332.67857142857144 days to get to Mars.  
Space Shuttle would take 0.5683628571428571 days to get to the Moon.
```

27.5.4 Part 4 - Create a Spacecraft Class

Organize the function and variables previously defined into a class named `Spacecraft`.

1. Fork the [part 4 repl.it](#).
2. Define a class named `Spacecraft`.
 - Properties
 - `milesPerKilometer: number = 0.621;`
 - `name: string;`
 - `speedMph: number;`
 - Constructor
 - `name` is the first parameter and it MUST be a string.
 - `speedMph` is the second parameter and it MUST be a number.
 - Sets the class properties using `this.name` and `this.speedMph`.
3. Move the function `getDaysToLocation`, defined in Part 3, into the `Spacecraft` class.
 - Update the function to reference the class properties `this.milesPerKilometer` and `this.speedMph`.
4. Create an instance of the `Spacecraft` class.
 - `let spaceShuttle = new Spacecraft('Space Shuttle', 17500);`
5. Print out the days to Mars.
 - Use template literals, `${spaceShuttle.getDaysToLocation(kilometersToMars)}` and `${spaceShuttle.name}` .
6. Print out the days to the Moon.
 - Use template literals, `${spaceShuttle.getDaysToLocation(kilometersToTheMoon)}` and `${spaceShuttle.name}` .

Expected Output

```
Space Shuttle would take 332.67857142857144 days to get to Mars.  
Space Shuttle would take 0.5683628571428571 days to get to the Moon.
```

27.5.5 Part 5 - Export and Import the SpaceLocation Class

1. Fork the [part 5 repl.it](#).
2. In repl.it, add a new file named `SpaceLocation.ts`.
3. Paste in the below code to `SpaceLocation.ts`.
 - Notice the `export` keyword. That is what allows us to import it later.

```

1 export class
2
3
4
5
6   this
7   this
8
9

```

4. Add the function `printDaysToLocation` to the `Spacecraft` class.
 - Notice that it takes a parameter of type `SpaceLocation`.

```

1           `${this}      } would take ${this
2   ↪       } days to get to ${                  }.
3

```

5. Import `SpaceLocation` into `index.ts`.
 - Add `import { SpaceLocation } from './SpaceLocation';` to the top of `index.ts`.
6. Print out the days to Mars and the Moon.

```

1 let           new          'Space Shuttle' 17500
2           new          'Mars'
3           new          'the Moon'

```

Expected Output

```
Space Shuttle would take 332.67857142857144 days to get to Mars.
Space Shuttle would take 0.5683628571428571 days to get to the Moon.
```

27.6 Studio: TypeScript

Let's practice TypeScript by creating classes for rocket cargo calculations.

27.6.1 Requirements

1. Fork [the starter repl.it](#).
2. Create classes for `Astronaut`, `Cargo`, and `Rocket`. (Details below)
 - All classes should be defined in their own files.
3. Use new classes to run a simulation in `index.ts` file.

In the starter code, you will notice that an interface named `Payload` has been declared. This interface ensures that any class that implements it will have a `weightKg` property.

27.6.2 Classes

Define each of these classes in a separate files. Each class should be exported using `export`.

```
export class
  // properties and methods
```

As needed, the classes can be imported using `import`.

```
import      './Astronaut'
```

Astronaut Class

- Defined in `Astronaut.ts`
- Implements the `Payload` interface
- Properties
 - `weightKg` should be a number.
 - `name` should be a string.
- Constructor
 - Parameter `weightKg` should be a number.
 - Parameter `name` should be string.
 - Sets value of `this.weightKg` and `this.name`.

Cargo Class

- Defined in `Cargo.ts`
- Implements the `Payload` interface
- Properties
 - `weightKg` should be a number.
 - `material` should be a string.
- Constructor
 - Parameter `weightKg` should be a number.
 - Parameter `material` should be a string.
 - Sets value of `this.weightKg` and `this.material`

Rocket Class

- Defined in `Rocket.ts`
- Properties
 - `name` should be a string.
 - `totalCapacityKg` should be a number.
 - `cargoItems` should be an array of `Cargo` objects.
 - * Should be initialized to an empty array `[]`
 - `astronauts` should be an array of `Astronaut` objects.
 - * Should be initialized to an empty array `[]`
- Constructor
 - Parameter `name` should be a string.
 - Parameter `totalCapacityKg` should be a number.
 - Sets value of `this.name` and `this.totalCapacityKg`
- Methods
 - `sumWeight(items: Payload[]): number`
 - * Returns the sum of all `items` using each item's `weightKg` property
 - `currentWeightKg(): number`
 - * Uses `this.sumWeight` to return the combined weight of `this.astronauts` and `this.cargoItems`
 - `canAdd(item: Payload): boolean`
 - * Returns `true` if `this.currentWeightKg() + item.weightKg <= this.totalCapacityKg`
 - `addCargo(cargo: Cargo)`
 - * Uses `this.canAdd(item: Payload)` to see if it can be added
 - If `true`, adds `cargo` to `this.cargoItems` and returns `true`
 - If `false`, returns `false`
 - `addAstronaut(astronaut: Astronaut)`
 - * Uses `this.canAdd(item: Payload)` to see if it can be added
 - If `true`, adds `astronaut` to `this.astronauts` and returns `true`
 - If `false`, returns `false`

27.6.3 Simulation in `index.ts`

Paste the below code into `index.ts`.

```
1 import             './Astronaut'
2 import             './Cargo'
3 import             './Rocket'
4
5 let                 new           'Falcon 9'  7500
6
7 let
8   new      75  'Mae'
9   new      81  'Sally'
10  new     99  'Charles'
11
12
13 for let 0
14  let
15
16
17
18 let
19  new      3107.39  "Satellite"
20  new      1000.39  "Space Probe"
21  new      753    "Water"
22  new      541    "Food"
23  new      2107.39  "Tesla Roadster"
24
25
26 for let 0
27  let
28
29
30
31  'final cargo and astronaut weight:'
```

Expected Console Output

```
Mae true
Sally true
Charles true
Satellite true
Space Probe true
Water true
Food true
Tesla Roadster false
final cargo and astronaut weight: 5656.78
```

27.6.4 Submitting Your Work

In Canvas, open the TypeScript studio and click the “Submit” button. An input box will appear.

Copy the URL for your repl.it project and paste it into the box, then click “Submit” again.

CHAPTER
TWENTYEIGHT

ANGULAR, PART 1

28.1 Why Use JavaScript Libraries

We use libraries to make our lives as developers easier. We can focus on solving our specific problems if we don't have to spend time figuring out how information flows throughout our project.

When it comes to web based applications, there are two very different places code can exist: in the user's browser (front end) and on the host's server (back end).

For web app development, consider the front end as what the user interacts with and sees, while the back end contains the logic and manipulations that the user doesn't need to worry about. Similar to how an old mechanical clock works. The front end would be the face with the 12 numbers and the two moving hands. The user only needs the clock face to determine the time. The back end for the clock would be the various cogs, wheels, and power source.

In this chapter we will use Angular as our front-end JavaScript framework. Angular dictates how to structure our files, as well as how the information flows between them. It also contains a number of tools to help us build the part of the application users see in their web browsers.

Note

There are many other ways to create front-end web applications with JavaScript. Popular frameworks for JavaScript include React, Vue, Ember, and others.

Angular is a framework that breaks the overall project into smaller pieces, each with their own code and styling. Angular then combines all of the pieces to create the full web page.

Taking this modular based approach allows us to separate the individual pieces of our application so we can focus on them one at a time.

Through the next three chapters we will look at the basic building blocks of an Angular application.

28.2 Templates

Take a look at the homepage for [LaunchCode](#). The content includes text, images, a navigation bar, buttons, embedded videos, static and scrolling sponsor logos, and links. All of this content is carefully arranged.

28.2.1 Your Own Website

Imagine building your own website with a smaller set of features. You could include favorite movies or sports teams, cities where you have lived, schools attended, hobbies, etc. Using your *HTML* and *CSS* knowledge, you could construct an appealing layout with carefully selected and arranged tags (`<div>`, `<button>`, ``, etc.).

Now imagine you have to change one or more items on the page. Maybe you move and your school or team loyalty shifts. Updating the city, school logo, etc. means adjusting those items in the HTML, and if that information appears in other areas of your website, then you need to modify that code as well. Also, you need to consider any formatting changes that result from adding or removing content.

Whew! Refreshing your website rapidly gets tedious, especially if it contains lots of information or consists of multiple pages. If only there was a way to automatically update the content and quickly rearrange the layout for your site...

28.2.2 Templates are Frameworks

A **template** provides the general structure for a web page. It identifies where different elements get placed on the page, but it does NOT fill them with content. Think of a template as an outline for what we want the page to look like. No details yet, just defined spaces where information needs to be added.

Let's see how using a template makes our lives easier.

No Template

The code below builds a simple 3-column webpage. It defines the location for each heading, unordered list, and button. The CSS file (not shown) specifies the font, text size, colors, etc.

```
1  body
2    h1 Hello, Screen!  h1
3    div class "row list"
4      div class 'movie'
5        h4 Movies  h4
6        ul
7          li Hidden Figures  li
8          li The Princess Bride  li
9          li Ferris Bueller's Day Off  li
10         ul
11        button class 'movie' More  button
12      div
13    div class 'school'
14      h4 Education  h4
15      ul
16        li LaunchCode  li
17        li Monsters University  li
18        li My HS  li
19        ul
20        button class 'school' More  button
21      div
22    div class 'hobby'
23      h4 Hobbies  h4
24      ul
25        li Knitting  li
26        li Cycling  li
27        li Shark Rodeo  li
28        ul
29        button class 'hobby' More  button
30      div
31    div
32    hr
33    div class "links"
34      h2 Links  h2
35      a href "https://www.launchcode.org/" target "_blank" LaunchCode  a  br
```

(continues on next page)

(continued from previous page)

```

36     a href "https://www.webelements.com/" target "_blank" WebElements a
37     div
38     body

```

repl.it

We could drastically improve the appearance and content of the page by playing around with the tags, classes, styles and text. However, any change we want to make needs to be coded directly into the HTML and CSS files.

This quickly becomes inefficient, especially if changing the items involves multiple blocks of code.

A Better Way

Each section in a template contains one or more *blanks* where specific items need to be added. Separate JavaScript code sends data to the template to fill in these blanks, and this data can change based on a user's actions.

```

1  body
2    h1 {{mainHeading}} h1
3    div class "row list"
4      div class 'movie'
5        h4 Movies h4
6        ul {{movieTitles}} ul
7        button class 'movie' More button
8      div
9      div class 'school'
10     h4 Education h4
11     ul {{schoolNames}} ul
12     button class 'school' More button
13   div
14   div class 'hobby'
15     h4 Hobbies h4
16     ul {{hobbies}} ul
17     button class 'hobby' More button
18   div
19   div
20   hr
21   div class "links" {{headingAndLinkList}} div
22 body

```

This HTML looks similar to the previous example, but saves about 16 lines. It provides the same `<div></div>` structure but replaces some of the specific text between the tags with *placeholders*.

Each item listed inside `{{ }}` refers to data that will be passed into the template and automatically formatted. For example, the template converts `{{movieTitles}}` into a sequence of `` tags.

By defining our template in an even more general manner, we could replace the `h4`, `ul` and `button` structure with a single placeholder.

```

1  body
2    h1 {{mainHeading}} h1
3    div class "row list"
4      div class 'movie' {{movieContent}} div
5      div class 'school' {{schoolContent}} div
6      div class 'hobby' {{hobbyContent}} div
7      div
8      hr

```

(continues on next page)

(continued from previous page)

```
9   div class "links" {{linkContent}} div  
10  body
```

By using a template to build the website, changing the list of movies, schools, or hobbies involves altering something as simple as an array or object. After changing that data, the template does the tedious work of modifying the HTML. The list of movies would update automatically if we add “Up” to our `favoriteMovies` array, which then gets passed into `{movieContent}`. We do not need to worry about re-coding any of the tags.

Templates Support Dynamic Content

If we add a search box to our website, a user could enter *NASA images*, *giraffe gif*, *movie trailers*, or something else. We cannot know ahead of time what a user will request, but we want our website to be able to display any relevant results.

Besides making it easier to organize and display content, templates also allow us to create a *dynamic* page. This means that its appearance changes to fit new information. For example, we can define a grid for displaying photos in rows of 4 across the page. Whether the images are of giraffes, tractors, or balloons does not matter. The template sets the layout, and the code feeds in the data. If more photos are found, extra rows are produced on the page, but each row shows 4 images.

Templates must be used anytime we create a webpage that responds to a changing set of data, especially if that data is unknown to us.

28.2.3 Templates Provide Structure, Not Content

Templates allow us to decide where we want to display data on our webpage, even if we do not know beforehand what that data will be. Information pulled from forms, APIs, or user input will be formatted to fit within our design.

In the figure, the black outlines represent different structures defined by the template. Each structure governs a specific portion of the screen. As data gets fed into the template, the appearance of the page changes.

If no data is sent to a particular structure, that part of the screen remains empty because the space is still reserved. Other components of the page will work around that space.

28.2.4 Check Your Understanding

Question

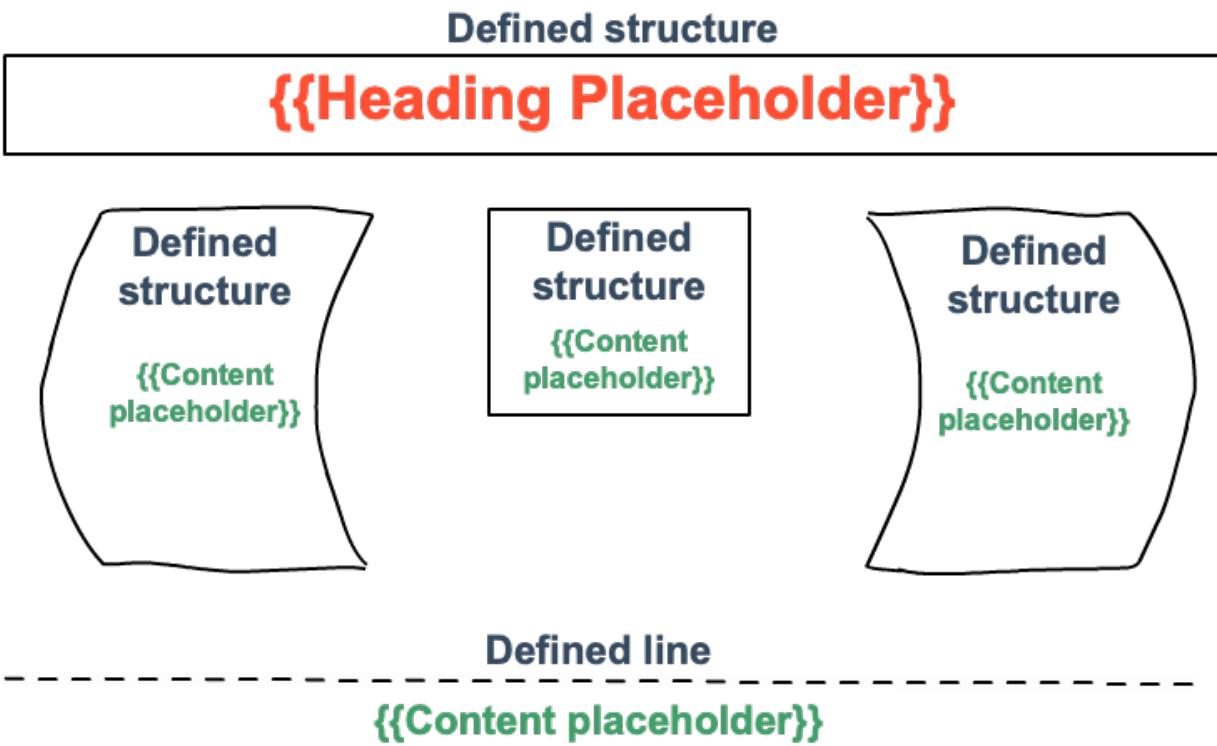
Why should we use a template to design a webpage rather than just coding the entire site with HTML and CSS?

Question

PREDICT: Do you think that changing the CSS for the *template* affects all of the smaller parts within that template?

1. Yes
 2. No
-

Question



PREDICT: Do you think that changing the CSS for one *component* in a template affects all of the other parts within that template?

1. Yes
 2. No
-

28.3 Angular File Structure

Each Angular project contains a standard file structure, which is shown in the figure below.

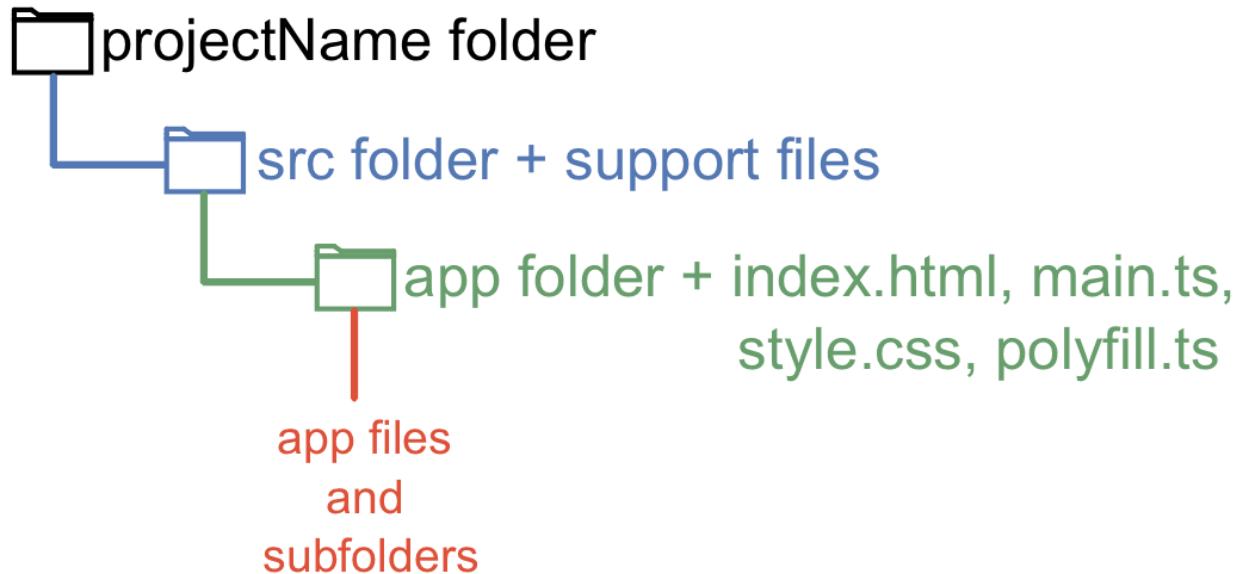
All of the project files (and there will be LOTS) are stored in the single, top-level folder, `projectName`, which contains a `src` folder and a set of support files.

Note

When you start a new Angular project, do not worry about the support files. These will be generated automatically, and they take care of the routine technical details for making your project run smoothly.

`src` contains the `app` folder and four important files: `index.html`, `main.ts`, `style.css`, and `polyfills.ts`. We will explore these files in more detail on the next page. For now, recognize that they control how the Angular project operates.

The `app` folder contains the files and subfolders needed to control the nitty-gritty details of displaying a webpage. For your projects, most of your time and effort will be spent modifying the contents within `app`.



28.3.1 StackBlitz

Your first Angular project will be another “Hello, World” example, and we will use a different online tool to build this. The site is called [StackBlitz](#), and it is similar to Repl.it in that it allows us to play without risk. The main difference is that StackBlitz is specifically designed to create webpage templates.

Note

For the “HTML Me Something” assignment, you created an account on [GitHub](#). Since you will modify an Angular project over the next few studios, you need access to GitHub to store your progress.

Follow the link and make sure you remember your login information, or enroll if you have not yet created an account.

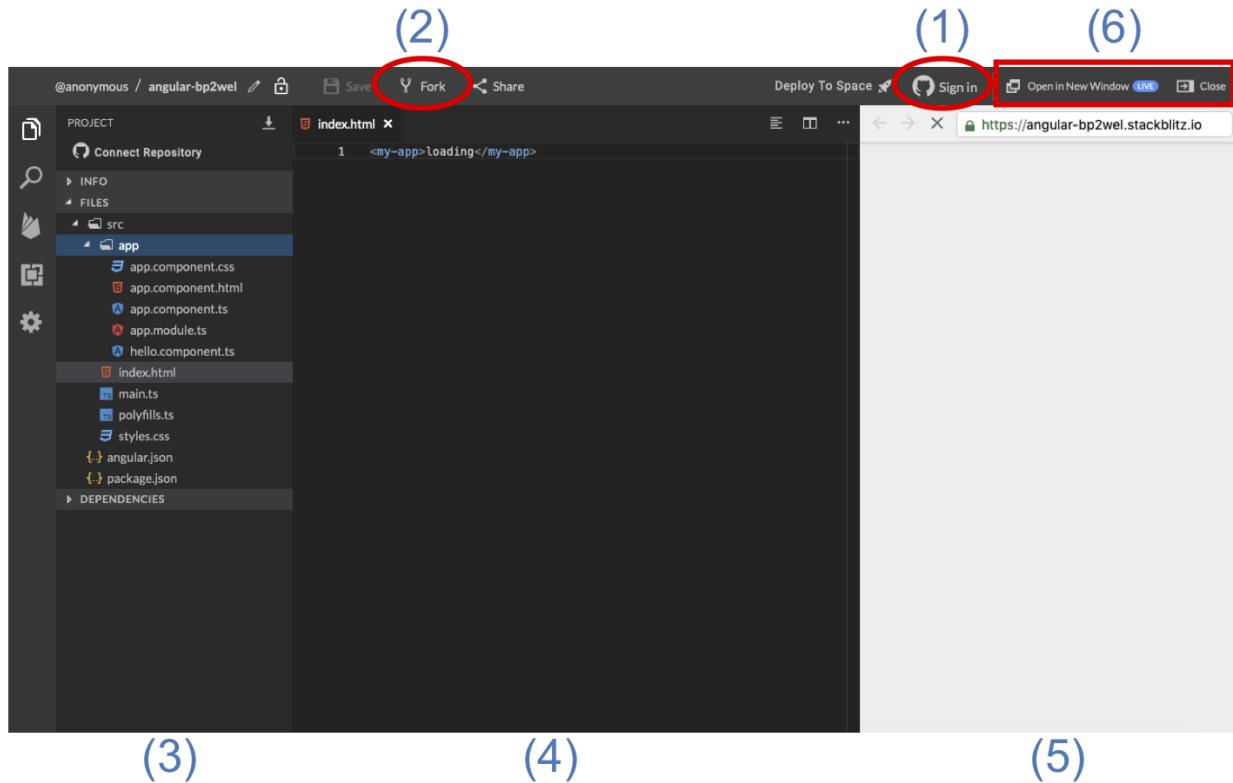
StackBlitz Workspace Layout

Before we dive into the project, we need to become familiar with the StackBlitz layout, which is based on the popular editor [Visual Studio Code](#).

The workspace consists of 3 main panels and several menu functions.

Features to note:

1. Use your GitHub account to store your projects. Click the GitHub button (1) to save your progress.
2. If you are viewing someone else’s project, you can *fork* the content (2) and store a copy of that project in your GitHub account. This allows you to edit the files without changing the originals, and it lets you use other programmers’ work (with permission) to enhance your own. As we learn more about Angular, you will need to fork sample code for the exercises and studios.
3. File panel and menu (3). Allows you to search for an item, add extensions, update settings, and add, open, or delete files.
4. Editor panel (4). Your code goes here. Click on a filename to open it in a tab in the editor.
5. Preview panel (5). Provides a view of what the webpage looks like. This panel can be minimized to save space or opened into a new browser window (6).



As with any new coding skill or tool, the best way to learn is to actively practice. Let's begin building your first Angular project.

28.4 Angular First Steps

The goal here is to create a minimum working webpage template. It will serve as a jumping off point for your later exercises and studios, so be sure you are logged into your GitHub account.

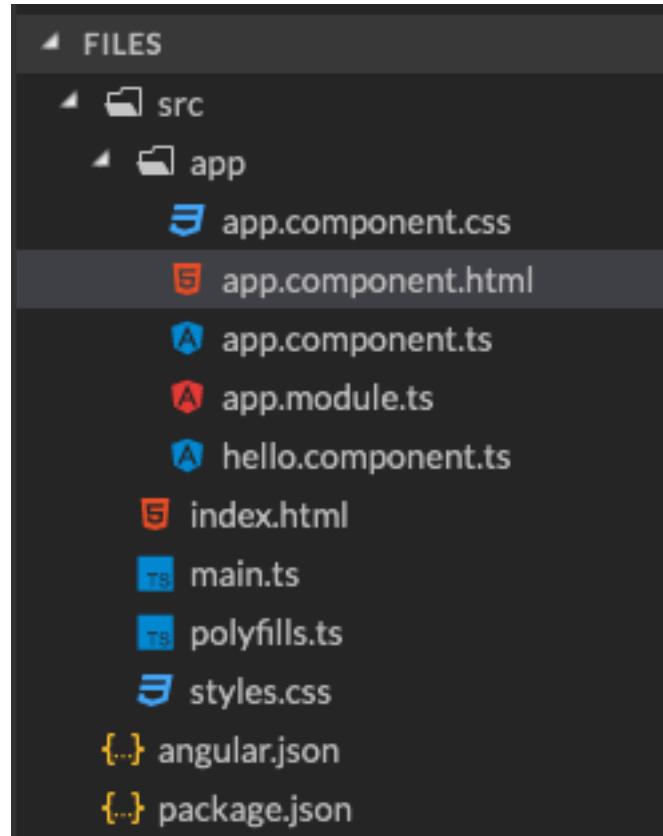
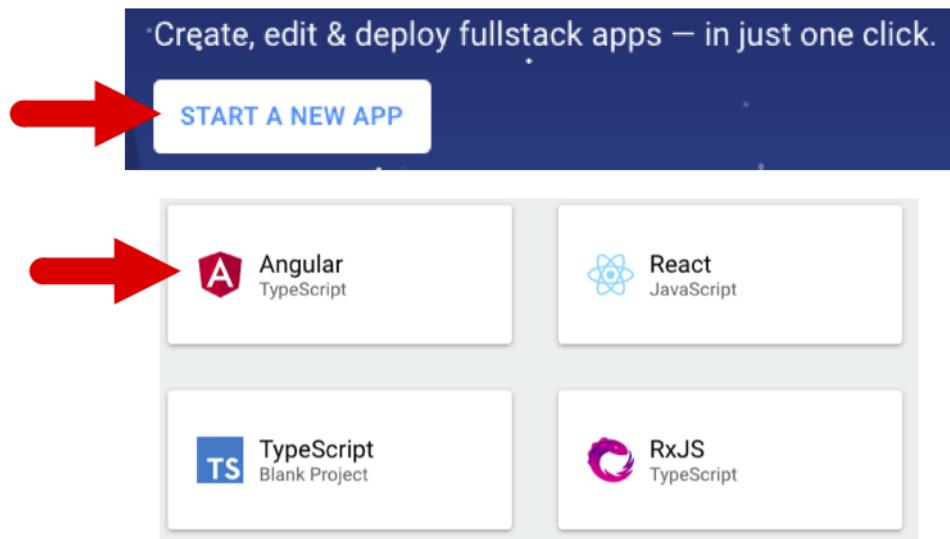
28.4.1 Starting a New Project

First, navigate to the [StackBlitz homepage](#).

Next, click “Start New App”, then select the “Angular (TypeScript)” option.

Examine the Files Created

1. The `src` folder holds the files and source code needed for the project.
2. The `app` folder holds the content for the webpage. Although the page is treated as a single entity, it actually consists of multiple pieces. `app` holds these different parts and establishes links between them. We will modify some of these files soon.
3. `index.html` is the highest level for displaying content. Anything added to this HTML file will appear on every page within a website.



4. `main.ts` imports the core methods required to make everything work. It also imports the content from the `app` folder.
5. `styles.css` holds the global style settings for the entire website.

28.4.2 Yay! A Webpage!

For a better view, go ahead and close the preview panel, then click the button to open that preview in a new browser window. You should see some simple content.

Hello Angular!

Start editing to see some magic happen :)

Congratulations! You have a functioning webpage. Feel free to play around a little bit before continuing, and do not worry about breaking anything. If necessary, you can always start another new project.

Try It!

Examine the files within the `app` folder. Modify the code to accomplish the following:

1. Add a nose to the emoticon.
2. Find where “Angular” is assigned to the heading, and then replace it with your name.
3. Change the color for the *Start editing...* sentence.

You may need to refresh the preview page after each attempt to see if any changes occur.

Now let’s take a look at the different project files.

What To Ignore

For every new project, StackBlitz includes the practice file `hello.component.ts`. It plays a role in generating the “Hello Angular” heading, but it does not do anything other than that. For our discussion here, we will ignore the file.

Warning

Do NOT delete `hello.component.ts` yet. There are some lines of code in `app.module.ts` that depend on this file. For now, leave `hello.component.ts` in place.

StackBlitz simplifies project creation by hiding many of the support files, and Angular itself automatically sets up the code to make the different parts of a project communicate with each other. `main.ts` and `polyfills.ts` are part of this automated process, so you should leave these files alone.

28.4.3 Inside the app folder

If we open `app.component.css`, we see three lines of code that styles the `<p>` tag. We can freely modify this file, but any CSS instructions only affect the HTML files within `app`. Also, the code in `app.component.css` overrides the CSS found in the higher level `styles.css` file.

This is a pattern for Angular. CSS instructions further down in the file tree have higher priority. If `app` contained a subfolder with its own `.css` file, then those instructions would be applied to the HTML files within that subfolder.

Let's examine the code contained in other three `app` files.

`app.component.html` File

Example

`app.component.html`

```
1 hello name "{{ name }}"    hello
2 P
3     Start editing to see some magic happen :)
4 P
```

`app.component.html` contains the structure and some of the text seen on the “Hello Angular!” page. Note that the file contains a placeholder, `{{ name }}`, that will be filled with data passed in from another file.

The strange tag `<hello>` represents a key idea behind building templates. Angular allows us to define our own tags, which are also used as placeholders in an HTML file. In this case, `<hello>` reserves space on the webpage for information supplied by the `hello.components.ts` file.

As we add more pieces to our template, we will define specific tags to help us arrange the different items on the screen. This makes it easier for us to keep track of our content. For example, if we want to build a webpage that contains a shopping list, a movies to watch list, and family photos, we can define the tags `<movies>`, `<grocery-list>`, and `<family-photos>`. With these tags, we can reference specific content whenever we want and clearly place it on a page. The tags also make it easy to play with new styles and formats for our grocery list without changing much code or altering the appearance of the movie list or photos.

`app.component.ts` File

Example

`app.component.ts`

```
1 import           '@angular/core'
2
3 @Component({
4   selector: 'my-app',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class
9   'Angular'
```

`app.component.ts` performs several important functions with very few lines.

1. Line 4 defines the tag `<my-app>`, which we can use in files that have imported `AppComponent`.
2. Line 5 imports `app.component.html`, which we examined above.
3. Line 6 imports `app.component.css`, which applies styling to the HTML file. (For those of you who changed the color of the *Start editing...* sentence in the Try It challenge above, this is why changing the css file worked).
4. Line 8 makes the styled `.html` file and anything defined in the `AppComponent` class available to other files.

Take a look at `app.component.html` again. We mentioned the `{} name {}` placeholder earlier and said that it gets filled with data from a different file. Line 9 in `app.component.ts` supplies this data by assigning the value 'Angular' to the `name` variable. Changing 'Angular' to a different value alters the webpage.

app.module.ts File

Example

`app.module.ts`

```
1 import           '@angular/core'
2 import           '@angular/platform-browser'
3 import           '@angular/forms'
4
5 import           './app.component'
6 import           './hello.component'
7
8 @NgModule
9
10
11
12
13 export class
```

Just like before, there is a lot going on within very few lines.

1. Lines 1 - 3 and line 9 import and assign the core modules that make Angular work. This is part of the automatic process, so do not play with these (yet).
2. Lines 5 and 6 import the classes `AppComponent` and `HelloComponent` from two local files, `app.component.ts` and `hello.component.ts`.
3. Lines 5 and 6 also pull in references to any other files linked to `app.component.ts` and `hello.component.ts`.
4. Line 10 declares the imported local files as necessary for the project.
5. Line 13 exports the `AppModule` class and makes it available to other files.

`app.module.ts` does the main work of pulling in the core libraries and local files. As new parts are added to a project, the import statements, `imports` array, and `declarations` array update automatically. We do not have to worry about the details for adding this critical code ourselves.

Note

Lines 6 and 10 are the reason why we cannot just delete the `hello.component.ts` file. Line 6 tries to import it, and line 10 says that the `HelloComponent` class defined in the file is needed.

28.4.4 Change The Content

Enough detail. Let's explore some more.

If you did not complete all of the *Try It* tasks above, attempt them now. After that...

Try It!

1. Replace line 1 in `app.component.html` with `<h1>{{name}}'s First Angular Project</h1>`.
 2. Define a variable in the `AppComponent` class to hold an array. Display the array items in an unordered list in the HTML file. Be sure to use placeholders.
 3. Define a rectangle object in `AppComponent` that has keys of `length`, `width` and `area`. Assign numbers to `length` and `width`, and have `area` be a function that calculates and returns the area.
 4. Use a `<p>` tag in the html file to display the sentence, “The rectangle has a length of ___ cm, a width of ___ cm, and an area of ___ cm².” Use placeholders in place of the blanks so the webpage displays the correct numbers.
-

28.4.5 Filename pattern

Many of the files we examined on this page contain the word `component` in the name. This results from the fundamental idea behind Angular. Each *template* for a webpage is constructed from smaller pieces, and these pieces are the *components*.

Our next step is to take a closer look at these building blocks within a template.

28.4.6 Check Your Understanding

Question

Where would be the BEST place to modify the code if we want a different font for `<p>` text within the template?

1. `app.component.ts`
 2. `app.component.html`
 3. `app.component.css`
 4. `app.module.ts`
-

Question

Where would be the BEST place to modify the code if we want to add a heading and an unordered list to the template?

1. `app.component.ts`
 2. `app.component.html`
 3. `app.component.css`
-

4. app.module.ts
-

Question

Where do we define a new HTML tag?

1. app.component.ts
 2. app.component.html
 3. app.component.css
 4. app.module.ts
-

28.5 Components

In Angular a **component** controls a patch of screen called a view.

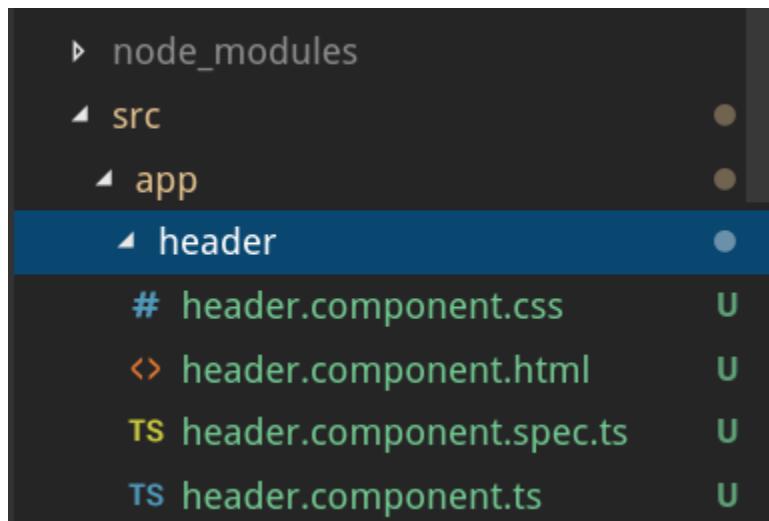
Angular builds a web page by combining multiple components together. Splitting our page into individual components makes our application more organized, and increases our ability to focus on one section of our web application at a time.

Everything in Angular centers on the idea of building a webpage from separate, smaller pieces. We must understand how to get these pieces to work together, and that begins by exploring what makes up each individual component. In order to build a reliable component, we must understand how each of its parts work and interact.

28.5.1 Component Files

A component consists of 4 files:

1. an HTML file (.html)
2. a CSS file (.css)
3. a typescript file (.ts)
4. a test file (.spec.ts)



Looking at the file tree, we see that all four files contain the name of the component. Also, the files are located in a folder named after the component. In the example above the component was named `header`.

If we add a new component named `task-list`, the four files created inside the `task-list` folder would be called:

1. `task-list.html`
2. `task-list.css`
3. `task-list.ts`
4. `task-list.spec.ts`

Each file contains the information necessary for ONLY that component. `task-list.html` holds the HTML required for the task-list and no other component. `task-list.css` only styles html within the `task-list` folder, the typescript code in `task-list.ts` only applies to this component, and all the tests for this component will be found in `task-list.spec.ts`.

28.5.2 Adding a New Component

Each component is a smaller part of an overall web application. The main component serves as a base structure, and it comes standard with all Angular applications. It's the container that holds all of the other components, and it organizes them into the web application.

An important thing to understand is how new components are created and then added to the main component.

When you generate a new component using a tool like Stackblitz or the Angular CLI, it is automatically added to the main component. Let's explore how this process works.

With the Angular CLI we generate a new component with the `ng generate component` command from the terminal.

```
[paul@paul mission-control]$ ng generate component task-list
CREATE src/app/task-list/task-list.component.css (0 bytes)
CREATE src/app/task-list/task-list.component.html (28 bytes)
CREATE src/app/task-list/task-list.component.spec.ts (643 bytes)
CREATE src/app/task-list/task-list.component.ts (280 bytes)
UPDATE src/app/app.module.ts (488 bytes)
```

From the output of the command we can see it creates four new files in the appropriate folder, and updates our `app.module.ts` file. Which results in the following file structure.

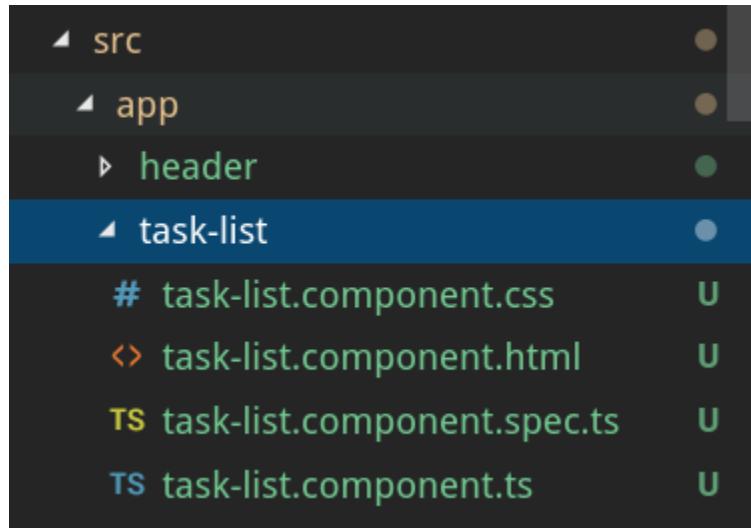
28.5.3 Component Nesting

Components can be put inside of other components. In essence, this is how the main component works. It is the component that holds all other components.

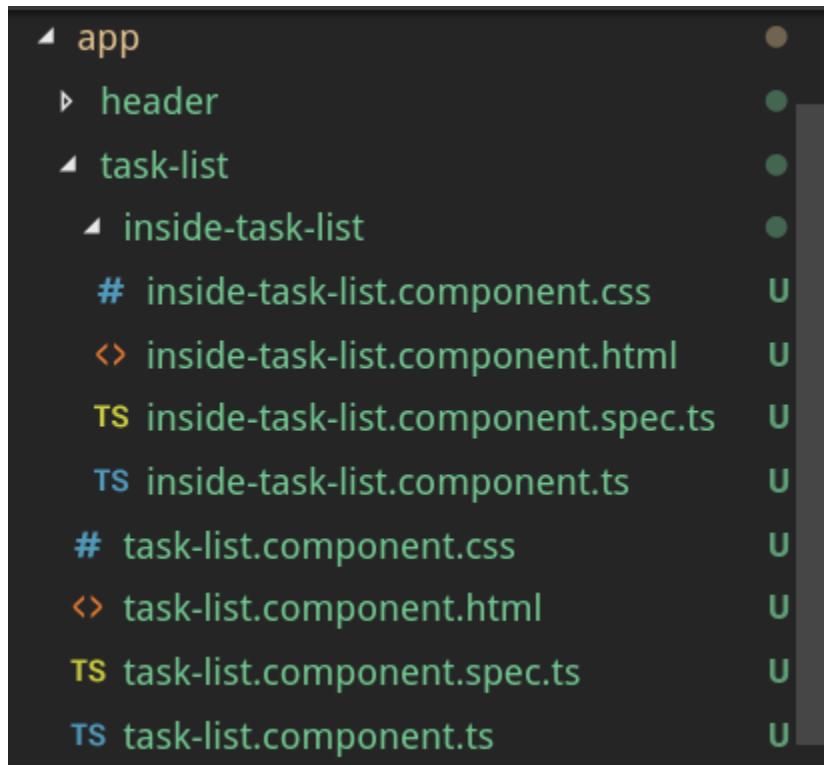
However, sometimes you may want to nest a new component inside of another one rather than in main.

Let's assume we want to add a new component within our `task-list` folder. In this case we simply change into the `task-list` directory from our terminal and then run the `ng generate component` command.

Running this command nests our new folder inside of the `task-list` folder, and it contains the four files we would expect.



```
[paul@paul mission-control]$ cd src/app/task-list/
[paul@paul task-list]$ ng generate component inside-task-list
CREATE src/app/task-list/inside-task-list/inside-task-list.component.css (0 bytes)
CREATE src/app/task-list/inside-task-list/inside-task-list.component.html (35 bytes)
CREATE src/app/task-list/inside-task-list/inside-task-list.component.spec.ts (686 bytes)
CREATE src/app/task-list/inside-task-list/inside-task-list.component.ts (307 bytes)
UPDATE src/app/app.module.ts (616 bytes)
[paul@paul task-list]$
```



When we nest a component inside of another, we still have all the files for the nested component at our disposal. Any CSS, HTML, or JavaScript we write will only affect the nested component and not the parent. However, the parent component DOES influence the nested one. For example, any CSS within `task-list.component.css` will apply to both `task-list.component.html` and `inside-task-list.component.html`. If we want `inside-task-list` to have different styling, we need to add code to the `inside-task-list` CSS file to override the parent.

28.6 Exercises: Angular, Lesson 1

28.6.1 Starter Code

For this set of exercises, follow the link to this [StackBlitz project](#) and click the *Fork* button. This allows you to edit the file.

`hello.component.ts` has been removed from the project, and two other components have been added.

Open the preview in a new window and examine the content of the webpage.

Note

There are some issues with running StackBlitz in the Firefox browser. The preview page may only be visible in a separate tab, some menu items may not be visible when logged into the site, and saving your progress becomes quirky.

For these exercises, we recommend using Safari or Chrome as alternates. If you insist on using Firefox, follow the instructions for *saving your work* carefully.

28.6.2 Part 1: Modify the CSS

The `movie-list` and `chores-list` components have been created, but so far they appear pretty bland. Let's change that.

1. Change the movie list text by adjusting the code in `movie-list.component.css` to accomplish the following:
 1. The text can be any color EXCEPT black.
 2. The movie list should have a centered heading.
 3. There should be at least 4 movie titles in an ordered list. Find the `movies` array in `MovieListComponent` and add more titles, then modify `movies.component.html` to display all the array entries.
 4. The font size should be large enough to easily read.
2. Change the chore list text by adjusting the code in `chores-list.component.css` to accomplish the following:
 1. Use a different font, with a size large enough to easily read.
 2. The text color should be different from the movie list, and not black.
 3. The chores list should have an underlined heading.
 4. The chores in the list should be italicized.

Complete the `fav-photos` Component

3. The `fav-photos` component has been generated, but it is incomplete. The page needs more images, which also need to be smaller in size.
 1. In the `FavPhotosComponent` class, assign a better section heading to `photosTitle`.
 2. The `image` variables should hold URLs for images, but only one is filled in. Complete at least one more, which can be from the web or personal pictures.
 3. In the `.html` file for this component, use placeholders and `img` tags to display one image per line.
 4. Use the `.css` file for this component to make the images all be the same size on the page.
 5. Refresh the preview page to see the updated content.

Before moving on, click the *Sign in* button. Save your work and rename the project (something like “angular-lesson-1-exercises”). This will allow you to store your progress and return to it later, if necessary.

28.6.3 Part 2: Add More Components

Note

You will be adding and modifying HTML elements for this project. If you need to review this topic, look back at the [HTML Tags](#) page, or try [W3Schools](#).

4. The page needs a title.
 1. Right-click the `app` folder in the files pane and select *Angular Generator/Component*. Name the new component `page-title`.
 2. Examine `page-title.component.ts` and note that the `app-page-title` tag has been defined next to `selector`. Shorten the tag name to just `page-title`.
 3. In the `PageTitleComponent` class, define a `title` variable and assign it a string.
 4. Add an `<h1>` to the `page-title.component.html` file. Use `{ {title} }` as a placeholder for the title you defined. Style the text to be underlined and centered on the screen.
 5. Add `<page-title></page-title>` to `app.component.html` and refresh the preview page to see your new content.
5. The page needs a set of links to favorite websites.
 1. Repeat steps 4a and b to generate a `fav-links` component.
 2. In the `FavLinksComponent` class, define a variable and assign it an array that contains two or more URLs.
 3. In the `.html` file for this component, use placeholders in a set of `<a>` tags for the web links. Each link should be on its own line.
 4. Add `<fav-links></fav-links>` to `app.component.html` and refresh the preview page to see the new content.

Note

Opening the `app.module.ts` file shows that the components for the movies, chores, title, links, and photos have all been automatically imported and declared.

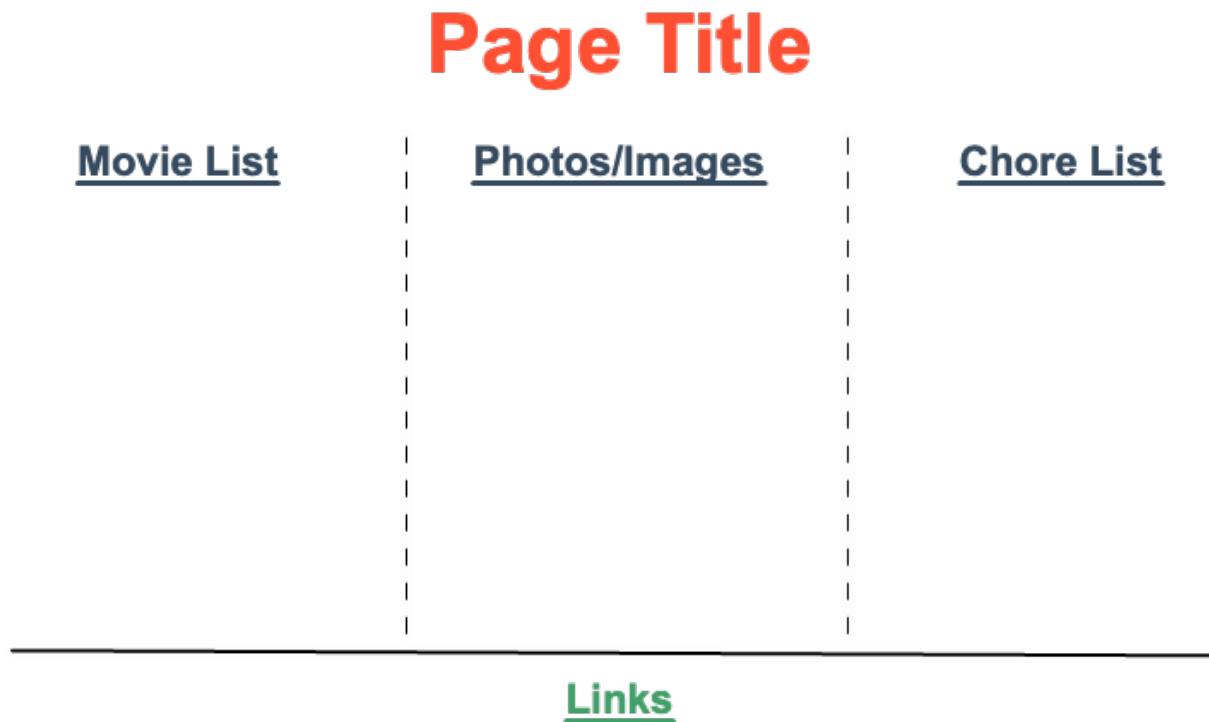
Whether or not we use StackBlitz, Angular automatically takes care of updating `app.module.ts` when new components are generated. However, *deleting* a component does NOT remove the references from the file.

28.6.4 Part 3: Rearrange the Components

The content on the page might appear a bit jumbled, since we gave you no guidance on where to add the custom tags in `app.component.html`. Fortunately, templates allow us to easily move items around the framework.

6. Rearrange the tags `fav-photos`, `fav-links`, `page-title`, etc. to create a specific page layout.
 1. `app.component.html` has `<div>` tags to set up a three-column row. Use this to arrange the movie list, images, and chore list.
 2. Center the title at the top of the page.
 3. Add a horizontal line below the three lists with the `<hr>` tag.
 4. Center the links below the horizontal line.

Your final page should have this format (the dashed lines are optional):



Optional Final Touches

7. To boost your practice, complete one or more of the following:
 1. Change the background to a decent color, image or pattern.
 2. Add a border around one or more of the components on the page.
 3. Add a fun, coding related gif to the page.

4. Make one component change when the user clicks on it.

28.6.5 Saving Your Work

As mentioned above, we recommend using Safari or Chrome for these exercises. The following instructions apply ONLY if you run StackBlitz in Firefox.

To save your work using Firefox:

1. Once you open the starter code, immediately click the *Fork* button. Do NOT login to StackBlitz yet.
2. Make the edits described in the exercises, saving your work as you go.
3. Once ALL of the exercises are complete, click the *Sign in* button and enter your password.
4. Rename the file by clicking the pencil icon near its name.

To modify your saved project using Firefox:

1. Click on the *Sign in* icon and enter your password.
2. Click on your username button to see a list of saved projects.
3. Copy the URL for your project (stackblitz.com/edit/my-project-name).
4. *Log out* of StackBlitz.
5. Paste the URL into the address bar and fork the project again.
6. Once you finish your new edits, log back in to StackBlitz and rename the file.

28.7 Studio: Angular, Part 1

Angular allows us to build webpages in small pieces, which keeps us organized and allows us to focus on one thing at a time.

In this chapter, we learned about the Angular file structure, templates, and components. Let's put these concepts into practice. Over the next three classes, we will build a Mission Planning Dashboard using our Angular skills.

Throughout the next three classes we will be building a Mission Planning Dashboard to practice our Angular skills.

28.7.1 Mission Planning Dashboard

A useful and common front end application is a *dashboard*. It shows quick information about a topic, so individuals using the web app can be well informed and make good decisions.

We will create a Space Mission Planning Dashboard.

This Angular project needs to include five sections:

1. A header that displays the mission and rocket name,
2. A list that displays the names of the crew,
3. A list that displays crew member roles,
4. A list that displays required equipment,
5. A list that displays the countries represented by the crew.

Each section needs to be its own component, and they all need to be placed within the `app` folder.

In the next two studios we will learn how to expand this project by using more Angular tools that will make the application more interactive. For the purposes of today's studio our web application will not be interactive, and will simply display static information.

28.7.2 Project Data

Data you will need for this project:

1. Mission name: Apollo 11
2. Rocket name: Saturn V
3. Crew member names & roles: Commander Neil Armstrong, Command Module Pilot Michael Collins, Lunar Module Pilot Buzz Aldrin
4. Equipment: Command Module, Lunar Module, 3 flight suits, 2 moon suits, 30 meals, 100 litres of water
5. Countries: United States

If you need a reminder on how to create components and organize them using the `app` component work through the :ref: “ again.

28.7.3 Bonus Mission

Add some CSS to change add different colors, fonts, borders, etc. to your dashboard.

Add links to information pages about the different countries represented by the crew.

Be creative! Make your webpage look good.

CHAPTER
TWENTYNINE

ANGULAR, PART 2

29.1 Angular Directives

In the *DOM chapter* you learned how to wait for *events* and then change the appearance of the webpage in response. You practiced these skills in the *DOM studio*. By waiting for the users to click specific buttons, your code launched, guided, and landed the LaunchCode rocket.

Angular helps us manage website content and manipulate the DOM through the use of *directives*. These simplify DOM changes by giving us alternatives to `getElementById`, `addEventListener`, `innerHTML`, etc.

There are three types of Angular directives:

1. **Components:** These control how a set of data gets displayed within a template.
2. **Structural directives:** These change the layout of the DOM by adding or removing elements (`div`, `ul`, `a`, `li`, etc.).
3. **Attribute directives:** These change the appearance of a specific element within the DOM.

We learned how to generate and modify components in the *last chapter*. In this lesson, we will focus on how to use structural directives to enhance our work.

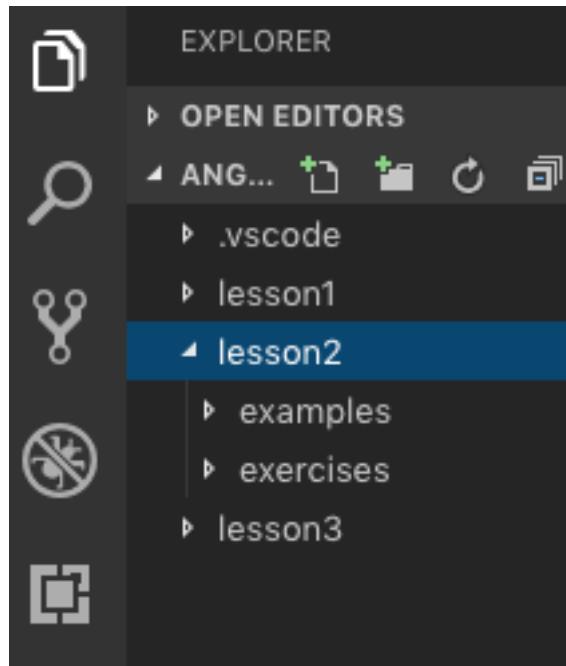
29.1.1 Open the Lesson 2 Folder

If you have not yet forked the Angular lessons repo, follow the directions given in *lesson 1*.

Open the `angular-lc101-projects` folder in VSCode and find `lesson2` in the sidebar.

Open the terminal panel and navigate to the `lesson2` folder. You should find subfolders for the examples and exercises used in this chapter. Also, completed code to some of the tasks is located in the `solutions` branch.

```
$ ls
    lesson1 lesson2 lesson3
$     lesson2
$ ls
    examples      exercises
$ git branch
    solutions
* master
```



29.2 ngFor

In the *Angular lesson 1 exercises*, you modified a movie-list component to display a series of titles. The final code within `movie-list.component.html` probably looked something like:

```
1  div class='movies'
2    h3 Movies to Watch  h3
3    ol
4      li {{movies[0]}}  li
5      li {{movies[1]}}  li
6      li {{movies[2]}}  li
7      li {{movies[3]}}  li
8    ol
9  div
```

`movies[0]` - `movies[3]` reference an array assigned within the `movie-list.component.ts` file.

To change the number of movie titles displayed in the ordered list, we could manually add or remove `li` tags, or we could use the structural directive `ngFor` to iterate through the movie options.

29.2.1 ngFor Syntax

The example below shows the basic approach for using `ngFor` to iterate through the contents of an array. For a more detailed guide to using `ngFor` and all of its variations, refer to the following resources:

1. [Angular documentation](#),
2. [Malcoded website](#).

Just like a `for` loop in JavaScript requires a specific syntax in order to operate, loops in Angular must follow a set of rules. Let's explore these rules by adding `ngFor` to our movie list code.

```

1  div class 'movies'
2      h3 Movies to Watch  h3
3      ol
4          li  ngFor "let movie of movies" {{movie}}  li
5          ol
6      div

```

Some items to note:

1. Structural directives all begin with the * symbol.
2. The string "let movie of movies" provides the instructions for running the loop.
 - a. The let keyword declares the movie variable.
 - b. of movies sets movie equal to the first element of the movies array. Each iteration of the loop sets movie equal to the next title in the array.
3. The *ngFor statement is placed INSIDE the tag.
4. {{movie}} is the placeholder for the current value of movie.

By placing the *ngFor statement inside the tag, the loop generates multiple elements. Each iteration adds a new list item to the HTML code, one for each title in the movies array.

Warning

The *ngFor statement generates a new HTML tag for each item in the array. *Be careful where you put the statement!* If we had added *ngFor = "let movie of movies" to the <h3> tag, then the Movies To Watch title would have been repeated multiple times.

In general, the syntax for *ngFor is:

```
*ngFor = "let variableName of arrayName"
```

Where variableName is the loop variable, and arrayName represents the array to iterate through.

Note

*ngFor only operates over the contents of an array. If we want to iterate over the characters in a string, we must first convert it into an array.

There is a technique for iterating over the key/value pairs of an object, but that is a more advanced topic. We will not discuss that method here.

29.2.2 Try It

From the lesson2 folder in VSCode, open the examples/ngfor-practice/src/app/chores folders and select the chores.component.html file.

The starter code should match this:

```

1  div class 'chores'
2      h3 Chores To Do Today  h3
3      ul
4          li  {{chores[0]}}  li

```

(continues on next page)

```
▲ lesson2
  ▲ examples
    ▲ ngfor-practice
      ▶ e2e
      ▶ node_modules
    ▲ src
      ▲ app
        ▲ chores
          # chores.component.css
          <> chores.component.html
          TS chores.component.spec.ts
          TS chores.component.ts
          ▶ movie-list
          # app.component.css
          <> app.component.html
          TS app.component.spec.ts
          TS app.component.ts
          TS app.module.ts
```



(continued from previous page)

```

5   li {{chores[1]}} li
6   li {{chores[2]}} li
7   ul
8   hr
9   div

```

In the VSCode terminal window, navigate to the `ngfor-practice` folder.

```

$ cd angular-lc101-projects/lesson2
$ ls
  examples      exercises
$ examples
$ ls
  input-practice  ngfor-practice  ngif-practice
$ ngfor-practice

```

Once you are in the folder, enter `npm install` in the terminal. This will add all of the Angular modules needed to run the project.

Enter `ng serve` to launch the project, then:

1. Modify `chores.component.html` with `*ngFor` to loop over the `chores` array:
 - a. Replace line 4 with `<li *ngfor = "let chore of chores">{{chore}}`.
 - b. Delete lines 5 and 6.
 - c. Save your changes.
 - d. Reload the webpage to verify that all the chores are displayed.
2. Open `chores.component.ts`. Add “Clean bathroom” to the `chores` array, then save. Reload the webpage to make sure the new chore appears. Your output should look like this:

Angular Lesson 2 Practice: ngFor

Movies to Watch

1. The Manchurian Candidate
2. Oceans 8
3. Hidden Figures
4. The Incredibles
5. The Princess Bride

Chores To Do Today

- Empty dishwasher
- Complete LaunchCode prep work
- Buy groceries
- Clean bathroom

3. Remove two chores from the array. Reload the webpage to make sure these items disappear from the list.
4. Use `*ngFor` within the `<div>` tag to loop over the `todoTitles` array:
 - a. Replace line 1 with `<div class='chores' *ngFor = "let title of todoTitles">`.
 - b. Replace “Chores To Do Today” in line 2 with a placeholder for `title`.
 - c. Save your changes, then reload the page. Properly done, your page should look something like:

LC101 Angular Lesson 2 Practice: ngFor

Movies to Watch

1. The Manchurian Candidate
2. Oceans 8
3. Hidden Figures
4. The Incredibles
5. The Princess Bride

Yesterday's Chores

- Empty dishwasher
- Complete LaunchCode prep work
- Buy groceries

Today's Chores

- Empty dishwasher
- Complete LaunchCode prep work
- Buy groceries

Tomorrow's Chores

- Empty dishwasher
- Complete LaunchCode prep work
- Buy groceries

5. Return to `chores.component.ts`. Add an item to the `todoTitles` array, then save. Check to make sure another list appears on the webpage. Next, remove two items from the `todoTitles` array. Save and make sure the page reflects the changes.

What If

1. What if you placed the `*ngFor` statement inside the `<h3>` tag instead of the `<div>` tag? Try it and see what happens!
2. What if you placed the statement inside the `` tag instead? Try it!

Bonus What If

What if we want to have different chores listed for Yesterday, Today, and Tomorrow?

Accomplishing this task is OPTIONAL, but it boosts your skill level and makes your page look better.

1. In the `chores.component.ts` file, replace the `chores` and `todoTitles` arrays with the following array of *objects*:

```
1      "Yesterday's Chores"      'Empty dishwasher'  'Start LaunchCode'
2      ↵prep work'  'Buy groceries'
3      "Today's Chores"        'Load dishwasher'   'Finish LaunchCode prep
4      ↵work'  'Buy the groceries you forgot'
          "Tomorrow's Chores"    'Empty dishwasher AGAIN'  'Play with
          ↵LaunchCode practice code'  'Groceries AGAIN'
```

(continues on next page)

LC101 Angular Lesson 2 Practice: ngFor

Movies to Watch

1. The Manchurian Candidate
2. Oceans 8
3. Hidden Figures
4. The Incredibles
5. The Princess Bride

Yesterday's Chores

- Empty dishwasher
- Start LaunchCode prep work
- Buy groceries

Today's Chores

- Load dishwasher
- Finish LaunchCode prep work
- Buy the groceries you forgot

Tomorrow's Chores

- Empty dishwasher AGAIN
- Play with LaunchCode practice code
- Groceries, AGAIN

(continued from previous page)

5

2. Update line 1 in chores.component.html to access each *object* in the chores array:
 - a. <div class='chores' *ngFor = 'let list of chores'>
 - b. Each iteration, list will be assigned a new object with title and tasks properties.
3. Update the placeholder in line 2 to access the title property of list.
4. Update line 4 to loop over the tasks array: <li *ngFor = 'let chore of list.tasks'>.

29.2.3 Check Your Understanding

The following questions refer to this code sample:

```

1  div
2    h3 My Pets  h3
3    ul
4      li {{pet}}  li
5    ul
6  div

```

Assume that we have defined a pets array that contains 4 animals.

Question

Adding *ngFor = 'let pet of pets' to the tag produces:

1. 4 headings
 2. 4 unordered lists
 3. 4 list items
 4. 4 headings each with 4 list items
-

Question

Moving `*ngFor = 'let pet of pets'` from the `` tag to the `<div>` tag produces:

1. 1 heading and 4 unordered lists with 4 pets each
 2. 4 headings and 4 unordered lists with 4 pets each
 3. 1 heading and 4 unordered lists with 1 pet each
 4. 4 headings and 4 unordered lists with 1 pet each
-

29.3 `ngIf`

The `*ngIf` structural directive evaluates a boolean expression and then adds or removes elements from the DOM based on the result.

The examples below show the basic usage of `*ngIf`. If you want to explore more details about this directive, refer to the following resources:

1. [Angular documentation.](#),
2. [Malcoded website.](#)

29.3.1 `*ngIf` Syntax

In general, the syntax for `*ngIf` is:

```
*ngIf = "condition"
```

`*ngIf` statements are added inside an HTML tag. The `condition` can either be a boolean or an expression that returns a boolean. If `condition` evaluates to `true`, then the HTML tag is added to the webpage, and the content gets displayed. If `condition` returns `false`, then the HTML tag is NOT generated, and the content stays off the webpage.

Note

`*ngIf` determines if content is *added or removed* from a page. This is different from determining if content should be *displayed or hidden*.

Hidden content still occupies space on a page and requires some amount of memory. *Removed* content is absent from the page—requiring neither space on the page nor memory. This is an important consideration when adding items like images or videos to your website.

Example

Let's modify our movie list code as follows:

```

1  div class 'movies' *ngIf "movies.length > 3"
2      h3 Movies to Watch h3
3      ol
4          li *ngFor "let movie of movies" {{movie}} li
5      ol
6  div

```

Adding the `*ngIf` statement inside the `<div>` tag determines whether that element and any content it contains gets added to the webpage. If the condition `movies.length > 3` evaluates to `true`, then the `div`, `h3`, `ol`, and `li` tags all get generated. If the condition returns `false`, then none of the tags are added to the HTML code.

Note

Only one structural directive can be assigned to an element. Since the `li` tag in the example above already contains `*ngFor`, we cannot add an `*ngIf` statement.

Logical Operators With `*ngIf`

Just like `if` statements, we can use the operators AND (`&&`), OR (`||`), and NOT (`!`) to modify the condition checked by `*ngIf`.

Examples

Logical AND:

```
p *ngIf "conditionA && conditionB" Some text p
```

Some text appears on the webpage only if `conditionA` and `conditionB` both return `true`.

Logical OR:

```
p *ngIf "conditionA || conditionB" Some text p
```

Some text appears on the page if either `conditionA` or `conditionB` return `true`.

Logical NOT:

```
p *ngIf "arrayName.length !== 0" Some text p
```

Some text appears when `arrayName.length` is NOT equal to 0.

What About `if/else`?

With JavaScript, we know how to use an `if/else` block to decide which set of code to execute:

```

1  if      5
2      //Execute this code if num is greater than 5.
3  else
4      //Execute this code if num is NOT greater than 5.
5

```

We can do the same thing in Angular to decide which set of HTML tags to generate. Unfortunately, setting this up with `*ngIf` is not as efficient.

The general syntax for adding an `else` block in Angular is:

```
1 someTag  ngIf  "condition; else variableName"
2   <!-- HTML tags and content --->
3   someTag
4
5 ng-template  variableName
6   <!-- Alternate HTML tags and content ---->
7   ng-template
```

Note that the `#` is required inside the `ng-template` tag.

Example

Let's modify the movie list example to create an alternate set of HTML tags if `movies.length > 3` returns `false`.

```
1 div class 'movies'  ngIf  "movies.length > 3; else needMoreMovies"
2   h3 Movies to Watch  h3
3   ol
4     li  ngFor  "let movie of movies"  {{movie}}  li
5   ol
6   div
7
8 ng-template  needMoreMovies
9   div class 'movies'
10    h3 Not Enough Movies!  h3
11    p You only have {{movies.length}} movies on your watch list!  p
12    p What's up with that?  p
13    p You need to purchase expensive popcorn more often.  p
14   div
15 ng-template
```

In line 1, the condition for `*ngIf` specifies what to do if `movies.length > 3` returns `true` or `false`:

1. If `true`, Angular executes lines 1 - 6.
2. If `false`, Angular searches for alternate code labeled with the name `needMoreMovies`. In this case, Lines 9 - 14 hold the alternate HTML tags.

`<ng-template></ng-template>` is a special Angular element. It contains content that *might* be required for a webpage. When the `else` statement in line 1 executes, Angular searches for an `ng-template` block labeled with the matching variable name. Angular then ignores the original `div` tags and anything they contain, and it replaces that content with lines 9 - 14.

29.3.2 Try It

In VSCode, return to your Angular Lesson 2 project. Use the terminal panel to commit any changes to your `*ngFor` work, then switch to the `ngif-practice` folder.

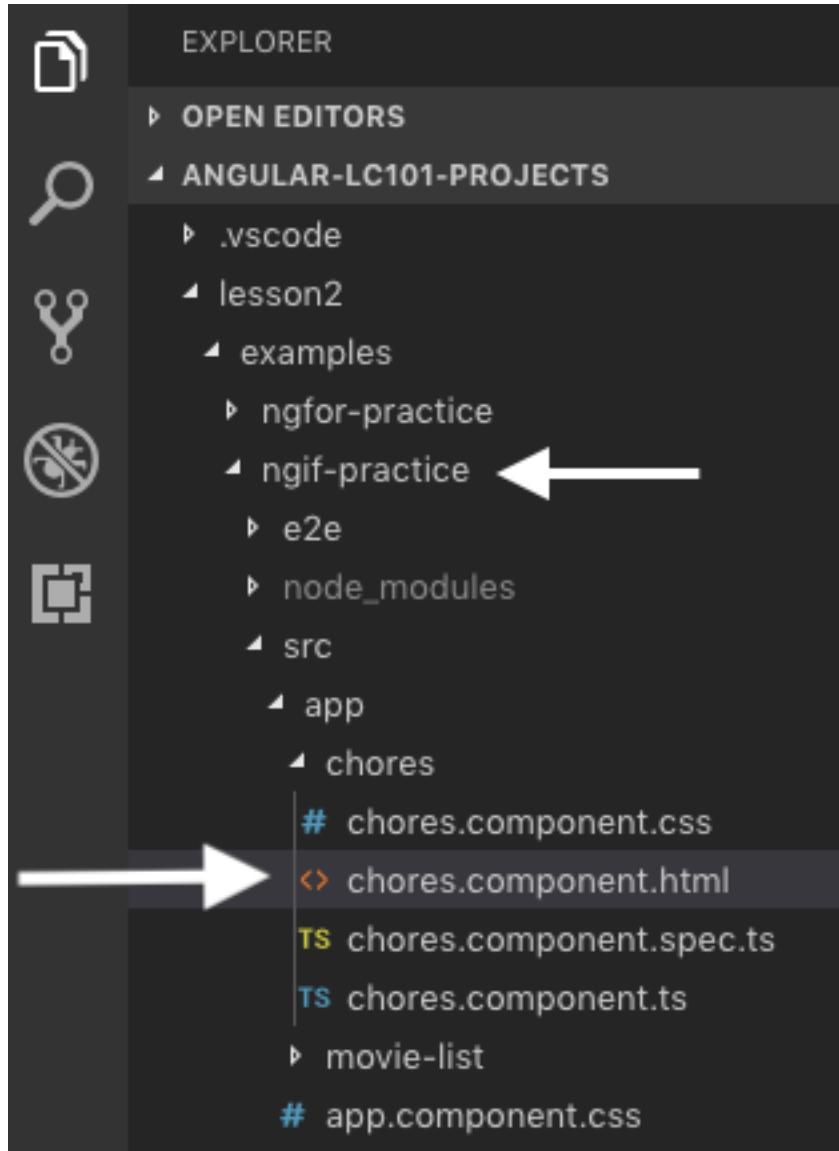
```
$ ..
$ ls
```

(continues on next page)

(continued from previous page)

```
input-practice  ngfor-practice  ngif-practice
$      ngif-practice
```

In VSCode, open the `chores.component.html` file from the `ngif-practice` folder:



The code should look like this:

```
1  div class="chores"
2    h3 Chores  h3
3    ul
4      li ngFor 'let chore of chores' {{chore}} li
5    ul
6  div
7  hr
8  div class="doneChores"
9    h3 Done Chores  h3
```

(continues on next page)

(continued from previous page)

```
10  ul
11    li  ngFor  'let done of finishedChores'  ((done))  li
12    ul
13  div
14  hr
```

Once again, you must install the necessary Angular modules. Run `npm install` in the terminal, then enter `ng serve` to launch the project.

Now use `*ngIf` to do the following:

1. Display the text “Work harder!” under the `Chores` list if the length of the `chores` array is longer than the length of the `finishedChores` array. Use a `p` tag for the text and make the words a different color.
2. Find the `chores` and `finishedChores` arrays in `chores.component.ts` and modify the number of items. Save your changes and reload the page to verify that your code works.
3. If the `chores` array is empty OR the `finishedChores` array has at least 3 more items than the `chores` array, display `trophyImage` under the `Done Chores` list. Otherwise, use a `p` tag to display the text, “No allowance yet.”
4. Return to `chores.component.ts` and change the number of items in the arrays again. Check to make sure the webpage correctly responds to your changes.
5. Finally, if the `chores` array is empty AND `finishedChores` contains 4 or more items, display an `h1` underneath the lists with the text “Ice cream treat!” Otherwise, display `h3` and `p` elements that describe how to earn ice cream.

Properly done, your page should look something like:

Angular Lesson 2 Practice: `ngIf`

Movies to Watch

- 1. The Manchurian Candidate
- 2. Oceans 8
- 3. Hidden Figures
- 4. The Incredibles
- 5. The Princess Bride

Chores

- Empty dishwasher

Done Chores

- Complete LaunchCode prep work
- Buy groceries
- Clean kitchen
- Call mom



To earn an ice cream treat:

The chores list must be empty.

The Done list must have at least 4 items.

29.3.3 Check Your Understanding

Examine the following code:

```
1  div
2    h3 Prep Work  h3
3    ul
4      li ngFor "let task of prepWork" (task) li
5    ul
6  div
7
8  ng-template noHW
9    p This space intentionally left blank.  p
10   ng-template
```

Question

Assume we have defined a `prepWork` array to hold the homework tasks for our next class. We want the webpage to always show the heading. Underneath that we want to add either the list of tasks or the paragraph text if the array is empty.

Where should we place an `*ngIf` statement to make this happen?

1. In the `div` tag
 2. In the `h3` tag
 3. In the `ul` tag
 4. In the `li` tag
 5. In the `ng-template` tag
 6. In the `p` tag
-

Question

For the same code sample, which of the following shows the correct syntax for the `*ngIf` statement?

1. `*ngIf = "prepWork.length === 0; else noHW"`
 2. `*ngIf = "prepWork.length !== 0; else noHW"`
 3. `*ngIf = "prepWork.length === 0; else #noHW"`
 4. `*ngIf = "prepWork.length !== 0; else #noHW"`
-

29.4 Events

Recall that *events* are actions that allow users to interact with the content on a webpage. Events include clicks, key presses, typing into an input box, hovering over images, etc.

Handling an event performs an action that influences the DOM.

Structural directives like `*ngFor` and `*ngIf` can be combined with events in order to add or remove content in response to user input. The image below shows an example of this idea.

When the user clicks “More” the hidden part of the text will be displayed. The layout of the page responds to the user’s actions. The click *event* is *handled* by showing more text.

29.4.1 Angular Events

In the *Event Listeners* section of the DOM chapter, we learned the syntax for using `addEventListener` and choosing the event we want.

Angular uses a different approach to listen for events. The event name is placed in parentheses () and added inside an HTML tag. This *binds* the event to that element.

The more common events include:

1. `(click)`: Waits for the user to click on the element.
2. `(keyup)`: Waits for the user to release a key.
3. `(keydown)`: Waits for the user to press a key.
4. `(mouseover)`: Waits for the user to move the cursor over the element.

Syntax

To bind an event to an HTML tag, the general syntax is:

```
<tag (event) = "statement"></tag>
```

The `statement` is the action we want to take when the event occurs. This could be a function call, a variable assignment, or just a value.

Examples

1. This code waits for the user to click the “Submit” button and then calls the `addData` function:

```
button click "addData(arguments)" Submit button
```

2. This code waits for the user to move the mouse over the element and then sets the `choice` variable equal to the value of `option`:

```
p mouseover "choice = option" {{option}} p
```

3. This code just waits for any key to be pressed:

```
div keydown "true" Press Any Key div
```

29.5 Responding to User Input

So far, if we want to change the number of items in our movie and chore lists, we have to go into the code and alter the arrays. This is inefficient. It would be better if we could make the changes on the screen.

29.5.1 Change Practice Folder

From the `lesson2` folder in VSCode, open the `examples/input-practice/src/app/movie-list` folders and select the `movie-list.component.html` file.

The starter code should match this:

```

1  div class='movies col-4'
2    h3 Movies to Watch  h3
3    ol
4      li ngFor "let movie of movies" {{movie}} li
5    ol
6    hr
7
8  div

```

In the terminal, navigate into the `input-practice` folder. Enter `npm install` to add the Angular modules, then run `ng serve`. The webpage should look like this:

Angular Lesson 2 Practice: User Input

Movies to Watch

1. Toy Story
2. The Shining
3. Sleepless in Seattle
4. The Martian

Now let's modify the code to accept user input. Be sure to code along with each step.

29.5.2 Setting Up an Input Box

Recall that the *HTML input tag* creates a box where the user can enter data. This data is retrieved by referencing the input element's `value` property.

1. Add `<input type='text' placeholder="Enter Movie Title Here"/>` to line 7, then reload the page.

Typing characters into the box does not do anything yet, since we have not included any instructions to deal with the data.

2. Angular extends the `input` element by declaring a *reference variable* and also an *event* that triggers data collection. Modify the `input` element in line 7 by adding `#newMovie` inside the tag:

```
input newMovie type 'text' placeholder "Enter Movie Title Here"
```

3. To store the typed characters in the `newMovie` variable, we need to wait for a particular event to occur. One of the simplest is to wait for a `keyup`. Modify line 7 one more time:

```
input newMovie keyup 'true' type 'text' placeholder "Enter Movie Title Here"
```

`(keyup)='true'` waits for the user to tap and release a key when the cursor is in the input box. When this happens, any characters in the box are stored in `newMovie`. However, we still need a place for the data to go.

4. Add a `<p>` element to line 8 of your code, and include a placeholder for the value of `newMovie`. Save and reload the page:

Movies to Watch

1. Toy Story
 2. The Shining
 3. Sleepless in Seattle
 4. The Martian
-

Enter Movie Title Here

```
1  div class='movies col-4'
2    h3 Movies to Watch  h3
3    ol
4      li  ngFor "let movie of movies" ({{movie}})  li
5      ol
6      hr
7      input newMovie  keyup 'true' type 'text' placeholder "Enter Movie Title Here"
8      ↵"
9      p {{newMovie.value}}  p
  div
```

When the keyup event is triggered, data gets stored in the newMovie variable. The newMovie.value placeholder on line 8 displays that data on the screen.

Try It

Save your work, reload the page, and then play! Type into the input box to verify your code works.

As you add or remove characters from the input box, you should see a real-time update of the text on the screen.

29.5.3 Changing the Event

In the example above, we used the keyup event to trigger data collection. This choice stores data in newMovie every time a key is released. For adding a new movie title to our list, however, it would be better to wait for the user to finish typing.

Modifying keyup

The keyup event waits for ANY key to be released, but Angular allows us to modify the keyword to wait for a SPECIFIC key. If, for example, we want to record data when the '0' key is released, then we would replace (keyup)='true' with (keyup.0)='true'.

Movies to Watch

1. Toy Story
 2. The Shining
 3. Sleepless in Seattle
 4. The Martian
-

Rutabaga!

Rutabaga!

The general syntax for this modification is `keyup.key`, and `key` refers to any character on the keyboard—including shift, enter, and space.

Try It

Change the `keyup` event in line 7 to:

1. `keyup.enter`,
2. `keyup.arrowup`,
3. `keyup.q`

Refresh the page after each change and explore how each one affects collecting user input.

Other key combinations are described at alligator.io.

Wait for a Click

5. Instead of waiting for a specific key to be pressed, let's wait for the user to click. Replace the `(keyup)` event with `(click)` in line 7:

```
input newMovie click 'true' type 'text' placeholder "Enter Movie Title Here"
```

Notice that the page now changes only when the user clicks inside the input box. This method is similar to `keyup.enter` because it gives the user a specific action to perform that records the entry without changing the text.

Now Add a Button

Since most of us are used to pressing the “Enter” key to submit our input, clicking inside the box might not be the best option. Fortunately, we know how to add a button to our HTML.

6. Add a <button> element with a click event to line 8. Also, change the event in line 7 back to keyup.enter:

```
1  div class='movies col-4'
2    h3 Movies to Watch  h3
3    ol
4      li  ngFor "let movie of movies" {{movie}}  li
5      ol
6      hr
7      input newMovie keyup.enter 'true' type 'text' placeholder "Enter Movie_"
8      ↵Title Here"
9      button click 'true' Add  button
10     p {{newMovie.value}}  p
11   div
```

Refresh the page and type a title into the input box. Tapping “Enter” or clicking the “Add” button should display your text below the box.

Movies to Watch

1. Toy Story
2. The Shining
3. Sleepless in Seattle
4. The Martian



29.5.4 Summary

We now have a way to collect user input and store it in a variable. We also learned how to access the data and display it on the webpage.

To accept user input, Angular requires three items:

1. The HTML input tag,
2. A variable to store the input, declared as #variableName,
3. An event that triggers data collection.

On the next page, we will learn how to make that input appear in our list of movies.

29.6 Events Can Call Functions

For the user input practice, we set the `keyup` and `click` events equal to `true`. This just checks to see if these events occur. When they do, the input is stored in `newMovie`, and the page refreshes.

To perform more complicated tasks in response to the user's actions, we can call a function when an event occurs. The syntax for this is:

```
(event) = 'functionName(parameters...)'
```

Changing the movie list displayed on the webpage requires us to modify the `movies` array in the `movie-list.component.ts` file. We will do this by creating an `addMovie` function and linking it to our event handlers.

29.6.1 Modify the HTML

Let's change our code in `movie-list.component.html` to call the function `addMovie` and pass the new movie title as the argument.

1. On lines 7 and 8, replace `true` with the function call:

```
1  div class='movies col-4'
2    h3 Movies to Watch  h3
3    ol
4      li ngFor "let movie of movies" {{movie}} li
5      ol
6      hr
7      input newMovie keyup.enter 'addMovie(newMovie.value)' type='text'_
→placeholder "Enter Movie Title Here"
8      button click 'addMovie(newMovie.value)' Add button
9      p {{newMovie.value}} p
10     div
```

Now when the user taps “Enter” or clicks the “Add” button after typing, the input `newMovie.value` gets sent to the function.

2. Since our plan is to use a function to add the new movie the array, we no longer need the title to appear below the input box. Remove `<p>{{newMovie.value}}</p>` from line 9.

29.6.2 Define the Function

Open `movie-list.component.ts` and examine the code:

```
1 import '@angular/core'
2
3 @Component({
4   selector: 'movie-list',
5   templateUrl: './movie-list.component.html',
6   styleUrls: ['./movie-list.component.css']
7 })
8 export class implements
9   'Toy Story' 'The Shining' 'Sleepless in Seattle' 'The Martian'
10
11 constructor
```

(continues on next page)

(continued from previous page)

14
15

The movies array stores the titles displayed on the webpage, and we want to update this when the user supplies new information.

3. Declare a function called addMovie that takes one parameter:

```
1  export class           implements
2      'Toy Story'  'The Shining'  'Sleepless in Seattle'  'The Martian'
3
4  constructor
5
6
7
8
9      : string
10
11
12
```

Notice that we have to declare the data type for the newTitle parameter.

4. Now add code to push the new title to the movies array:

```
1  export class           implements
2      'Toy Story'  'The Shining'  'Sleepless in Seattle'  'The Martian'
3
4  constructor
5
6
7
8
9      : string
10     this
11
12
```

The keyword this is required.

Note

It is a common practice to put constructor and functions like ngOnInit AFTER the variable declarations but BEFORE any custom functions.

Save the changes and then refresh the page. Enter a new title to verify that it appears in the movie list. Your page should look something like:

29.6.3 Tidying Up the Display

Notice that after adding a new movie to the list, the text remains in the input box. If we click “Add” multiple times in a row, we would see something like:

Let’s modify the code to try to prevent this from happening.

Movies to Watch

1. Toy Story
2. The Shining
3. Sleepless in Seattle
4. The Martian
5. Rutabaga

Rutabaga

Add

Movies to Watch

1. Toy Story
2. The Shining
3. Sleepless in Seattle
4. The Martian
5. Avengers: Endgame
6. Avengers: Endgame
7. Avengers: Endgame
8. Avengers: Endgame

Avengers: Endgame

Add

Clear the Input Box

5. After the user submits a new title, we can clear the input box by setting its value to be the empty string (' '). Open `movie-list.component.html` and modify the input statement as follows:

```
input newMovie keyup.enter "addMovie(newMovie.value); newMovie.value = ''"  
  type "text" placeholder "Enter Movie Title Here"
```

When `keyup.enter` occurs, the code calls `addMovie`. Once control returns from the function, `newMovie.value` is set equal to ' ', which clears any text from the input box.

6. Since the user can also click the “Add” button to submit a title, we need to modify the `<button>` element as well:

```
button click "addMovie(newMovie.value); newMovie.value = ''" Add button
```

Now `newMovie.value` is set equal to ' ', when “Enter” or “Add” are used to submit data.

Try It

Refresh the page and verify that the input box gets cleared after each new title.

Check for Duplicates

Even though we clear the input box, there is nothing to prevent the user from entering the same movie multiple times. While some fans may want to watch a film twenty times in a row, let’s have our code prevent repeats.

Recall that the *includes method* checks if an array contains a particular element. The method gives us several ways to check for a repeated title. One possibility is:

```
1      : string  
2  if this  
3    this  
4  
5
```

If the `movies` array already contains `newTitle`, then the `includes` method returns `true`. The NOT operator (!) flips the result to `false`, and line 3 is skipped.

Try It

Refresh the page and verify that you cannot enter a duplicate title.

29.6.4 Bonus

To boost your skills, try these optional tasks to enhance your work:

1. Modify `addMovie` to reject the empty string as a title.
2. Use `*ngIf` to display an error message if the user does not enter a title or submits a title that is already on the list.
3. Add CSS to change the color of the error message.

The `example-solutions` branch of the Angular repo shows completed code for the bonus tasks.

29.6.5 Check Your Understanding

Assume that we have an Angular project that presents users with a list of potential pets:

Potential Pets

1. Dalmatian
2. Tiger
3. Caterpillar
4. Tardigrade
5. Tortoise

Question

Which of the following calls the `addPet` function when the user clicks on one of the potential pets:

1. `{{pet}}`
 2. `<li (click) = "true">{{pet}}`
 3. `<li #addPet (click) = "true">{{pet}}`
 4. `<li (click) = "addPet(pet)">{{pet}}`
-

Question

When the user moves the mouse over an animal, we want to store its name in the `newFriend` variable. Which of the following accomplishes this?

1. `<li (mouseover) = "pet.name">{{pet}}`
 2. `<li #newFriend (mouseover) = "pet.name">{{pet}}`
 3. `<li (mouseover) = "newFriend = pet.name">{{pet}}`
 4. `<li (mouseover) = "newFriend">{{pet}}`
-

29.7 Exercises: Angular, Lesson 2

Let's build an interactive webpage that allows us to review data for our astronaut candidates and select crew members for a space mission.

29.7.1 Starter Code

The starter code for the exercises is in the same [repository](#) that you cloned for the chapter examples.

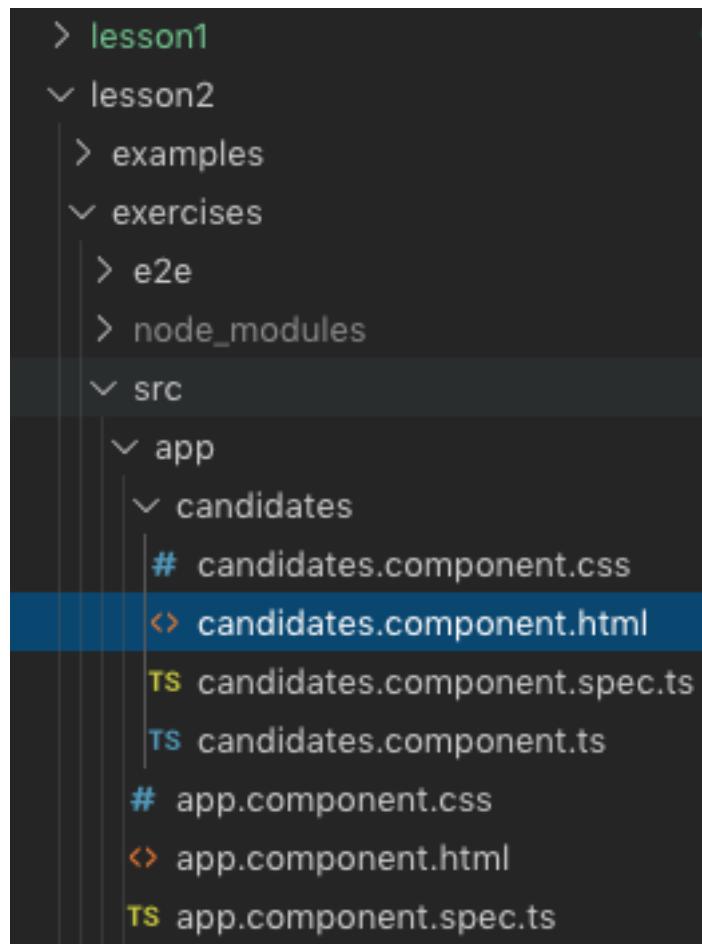
Note

Remember that the repository contains a `master` branch with all the starter code as well as a `solutions` branch showing the completed exercises.

The solutions provide a resource for you to check if you get stuck. However, for best results you should make a *valiant* attempt at solving the tasks before looking at “the answers”.

Also, if your code works but is different from the solutions, that is OK. There are usually multiple ways of solving the same problem.

From the `lesson2` folder in VSCode, navigate into the `exercises/src/app/candidates` folder. Open the `candidates.component.html` and `candidates.component.ts` files.



In the terminal, navigate into the lesson 2 `exercises` folder. Enter `npm install` to add the Angular modules, then run `ng serve`. When you open the webpage in your browser, it should look like this:

29.7.2 Candidates Column

Examine the `candidates` array in `candidates.component.ts`. It contains one object for each animal astronaut. We want to start by listing the names of the animals in the “Candidates” column of the webpage.

Exercises: Angular Lesson 2

Mission Name: LaunchCode Moonshot

Candidates	Candidate Data	Sidekick	Selected Crew
<p>1. Use *ngFor to automatically list all of the candidates' names.</p>	OOPS! No image available.		<ul style="list-style-type: none"> • Use *ngFor to list the selected crew.

[Clear Data & Image](#)

1. Find the “Candidates” section in `candidates.component.html`. Use `*ngFor` in the `` tag to loop over the `candidates` array and display each name in an ordered list.
2. We want each name to be interactive. Add a `click` event to the `` tag. When a user clicks on a name, set the variable `selected` to be equal to the chosen candidate.

Properly done, your output should behave something like this:

29.7.3 Candidate Data Column

When we click on a candidate’s name, we want their information to appear in the “Candidate Data” column. If no candidate is selected, we want the space under the heading to remain blank.

1. In the `<p></p>` element underneath the “Candidate Data” heading, add labels for a candidate’s Name, Age, Mass, and Sidekick.
2. Add placeholders to display the candidate’s data next to each label.
3. Use `*ngIf` inside the `<p>` tag to check if a candidate has been selected. If so, display the labels and the data.
4. Next, create a way to clear the data. In the `<button>` tag for “Clear Data & Image”, add a `click` event that sets `selected` to `false`.

Properly done, your output should behave something like this:

29.7.4 Sidekick Image Column

Every good hero needs a loyal sidekick, and our candidates are no exception!

When we click on a candidate’s name, we want an image of their sidekick to appear under the “Sidekick” column. If no candidate is selected, we want this area to remain blank.

1. In the `` tag, use `*ngIf` to check if a candidate has been selected.
2. Replace the generic `{{placeholder}}` with the `image` property of the candidate.

Properly done, your output should behave something like this:

29.7.5 Selected Crew Column

Once we select a candidate, we want an option to add them to the crew of the next space mission.

1. In `candidates.component.ts`, code an `addToCrew` function that takes an *object* as a parameter.
2. If the candidate is NOT part of the crew, the function should push them into the `crew` array. Candidates who are already part of the crew should be ignored.
3. In `candidates.component.html`, add a “Send on Mission” button next to the “Clear Data & Image” button.
4. Add a `click` event to the button to call the `addToCrew` function. When clicked, pass the selected candidate as the argument.
5. Under the “Selected Crew” section, use `*ngFor` to loop over the `crew` array and display each name.

Clear Crew List

1. Add a “Clear Crew List” button under the “Selected Crew” list.
2. This button should only appear when the `crew` array contains data. Use `*ngIf` to make this happen.
3. Add a `click` event that clears the `crew` array.

Properly done, your output should behave something like this:

29.7.6 Bonus Missions

Fine Tune the Buttons

1. Update the `Send on Mission` button to appear only if a candidate has been selected.
2. Make the `Send on Mission` button disappear if the selected candidate is already part of the crew.
3. Make the `Send on Mission` button disappear once three crew members have been assigned to the mission.

Change the Mission Name

We can make the Mission Name heading interactive. When clicked, we want to present the user with an input box to enter a new name.

1. Replace line 2 in `candidates.component.html` with `<h2 class="centered" *ngIf = "!editMissionName; else editMission" (click)="editMissionName = true">Mission Name: {{missionName}}</h2>`.
2. When clicked, the `ng-template` code executes. Update the `input` tag with a `keyup.enter` event. The event should call the `changeMissionName` function and pass the new name as an argument.
3. In `candidates.component.ts`, code a `changeMissionName` function to update the name of the mission.
4. After changing the mission name, set `editMissionName` to false.

29.7.7 Bonus Results

After finishing the bonus missions, your output should behave something like this:

29.8 Studio: Angular, Part 2

At the end of the first mission planner studio, multiple components display data about the mission. Your job is to allow the user to update the mission plan by adding user interaction.

29.8.1 Getting Started

This studio uses the same mission planner repository as Angular studio part 1.

1. Open the [mission planner repository](#) in VSCode
2. Run `git status` to see if you have any uncommitted work, if you do resolve it
3. Checkout the `studio-2` branch
4. Run `npm install` to download dependencies
5. Run `ng serve` to build and serve the project

29.8.2 Review the Starter Code

The starter code for this studio is similar to the *solution* for the first mission planner studio, but with a few notable changes.

Editable Mission Name

The mission name can now be edited by clicking on the text, changing the text in the input box, and then updated by clicking save or pressing the enter key. Review the code in `src/app/header/header.component.html` and `src/app/header/header.component.ts` to see how this feature was implemented.

Fig. 1: Example of mission name being edited.

Crew Array of Objects

Open `src/app/crew/crew.component.ts` in VSCode. Notice on line 10 that a crew array is defined. This array of objects will be used to display the crew. Each crew member has a name and `firstMission` property. If `firstMission` is true, it means this is the first mission for that person.

```
1 import           '@angular/core'  
2  
3 @Component  
4   'app-crew'  
5     './crew.component.html'  
6     './crew.component.css'  
7
```

(continues on next page)

(continued from previous page)

```
8  export class           implements
9
10   : object
11     "Eileen Collins"      : false
12     "Mae Jemison"        : false
13     "Ellen Ochoa"         : true
14
15
16   constructor
17
18
19
20
21
```

29.8.3 Requirements

Note

All of these features only *temporarily* alter the data. If you refresh the page, the original data will reappear.

Edit Rocket Name

The rocket name should be clickable and editable like the mission name. Alter `src/app/header/header.component.html` and `src/app/header/header.component.ts` to allow the user to edit the rocket name.

Use `*ngFor` to Display Crew

Replace the static list of `` tags in `src/app/crew/crew.component.html` with an `*ngFor` that loops over the `crew` array.

Add this code to `src/app/crew/crew.component.html`.

```
1  li  ngFor "let member of crew" {{member.name}} li
```

Display 1st Mission Status

If a crew member's `firstMission` property is `true`, then display the text “- 1st” next to their name.

Add Crew Members

Allow crew members to be added to the list. To create a new crew member, two pieces of information are required:

1. crew member's name
2. the first mission status

We will use an input box and a `checkbox` to collect the data.

Add this code to the *bottom* of `src/app/crew/crew.component.html`.

Crew

- Eileen Collins
- Mae Jemison
- Ellen Ochoa - 1st

Fig. 2: Example of first mission status being shown.

Fig. 3: Example of crew member being added.

```

1  input name type "text"
2  label First mission input firstMission type "checkbox"    label
3  button click "add(name.value, firstMission.checked)" Add button

```

Line 1 creates an input that declares the local variable `name`. Line 2 defines a checkbox that declares the `firstMission` variable. Line 3 creates a button that, when clicked, sends the new `name` and `checkbox` value to the `addCrewMember` function.

Add the below `add` function to the `crew` component in file `src/app/crew/crew.component.ts`.

```

1      : string          : boolean
2  this           : memberName        : isFirst
3

```

Remove Crew Members

Allow removing of crew members by adding a button next to each person in the crew list. When the remove button is clicked, the `remove` function in the `crew` component will be called which will delete that person from the `crew` array.

Fig. 4: Example of crew member being removed.

Add line 3 to file `src/app/crew/crew.component.ts`. Be sure to put it before the closing ``, so that the button appears next to each item in the crew list.

```

1  li ngFor "let member of crew"
2    {{member.name}}
3    button click "remove(member)" remove button
4  li

```

Add the below `remove` function to the `crew` component in file `src/app/crew/crew.component.ts`.

```

1      : object
2  let      this
3  this
4

```

Edit Crew Members

Finally we are going to allow the user to edit crew members who have already been added.

1. If the crew member name is clicked, then their name should be replaced with a text input and a save button.
2. When save is clicked the input and save button are replaced by the text only version of the name.
3. Only one crew member can be edited at a time.

Fig. 5: Example of crew member name being edited.

We need to add a click event to the member name.

4. Put `{member.name}` inside of a `` that has a `(click)` handler.
5. Make the `` in `src/app/crew/crew.component.html` look like the code below.

```
1 li ngFor "let member of crew"
2   span click "edit(member)" class "editable-text" {{member.name}} span
3   button click "remove(member)" remove button
4 li
```

We need a way of knowing which crew is being edited.

6. Add this property to the crew component in file `src/app/crew/crew.component.ts`. The property `memberBeingEdited` represents the crew member who is currently being edited.

```
: object null
```

7. Next add a `edit` function to the crew component file `src/app/crew/crew.component.ts`. This function will set a `memberBeingEdited` variable to be equal to the crew member who was clicked.

```
: object
this
```

Now we need to add an `*ngIf` that will show the two versions of the member, the display state or the edit state.

8. In the edit state an input box with a save button will appear, but for now the input and save won't have any functionality. Make your `src/app/crew/crew.component.html` file look like the below code.

```
1 h3 Crew h3
2 ul
3   li ngFor "let member of crew"
4
5     span ngIf "memberBeingEdited !== member; else elseBlock"
6       
7         span click "edit(member)" class "editable-text" {{member.name}} span
8         span ngIf "member.firstMission"
9           - 1st
10          span
11            button click "remove(member)" remove button
12          span
13
14     ng-template elseBlock
15       
16         input
```

(continues on next page)

(continued from previous page)

```

17     button save button
18     ng-template
19
20     li
21       ul
22       input name type "text"
23       label First mission input firstMission type "checkbox" label
24       button click "add(name.value, firstMission.checked)" Add button

```

Finally we are going to make the edit state update the member name when save is clicked.

9. Update the <input> and <button> tags to look like:

```

1   ng-template elseBlock
2     <!-- edit state of member -->
3       input updatedName keyup.enter "save(updatedName.value, member)" value
4       ↵"{{member.name}}"
5         button click "save(updatedName.value, member)" save button
6       ng-template

```

The last step is to add the `save` function to the crew component. This function will be called when the <button> is clicked or when the enter key is pressed and the <input> has focus.

10. Add the below `save` function to the crew component.

```

1   : string      : object
2   'name'        null
3   this
4

```

29.8.4 Bonus Missions

Before starting on any of these bonus features, be sure to commit and push your work.

1. Don't allow duplicate names to be added to the crew.
2. Allow user to add equipment.
3. Allow the user to edit equipment.
4. Allow the user to remove equipment.
5. Allow user to add experiments.
6. Allow the user to edit experiments.
7. Allow the user to remove experiments.

BOOSTER ROCKETS

30.1 Best Practices: Learning To Code

For years, the education community has debated the best approach for *homework*. Should it be scored or not? What is the proper amount? Should it be even be assigned? Many reports begin with “Studies have shown...,” but no universally accepted conclusion has emerged.

We cannot possibly restate all the conclusions here, nor will we present any part of the debates. Instead, here are a few simple facts for LC101:

1. Yes, there is homework, which consists of prep work, exercises, studios, and assignments.
2. Only the assignments count toward your final grade.
3. You do not have to do the ungraded homework, but you absolutely *SHOULD*.

The rest of this page is our attempt to motivate you to complete ALL of the work for LC101. How much encouragement you need depends on your personality. Read the sections that appeal to your psyche and are most likely to encourage you.

“Thanks, LaunchCode, but you convinced me at, ‘Do all the homework.’ ”

If this is your thinking, then bless you. Skip to the best practices listed in the last section of this page. *You’re awesome!*

Otherwise, on to the motivation...

30.1.1 Philosophy & An Appeal to Wisdom

I hear, and I forget. I see, and I remember. I do, and I understand. - Chinese Proverb

Experience is definitely the best teacher. You could read pages and pages about `for` loops, which will give you a handle on the vocabulary. However, until you actually construct your first working loop, your understanding will be incomplete.

Tip

You learn to code by coding. *Do your homework.*

30.1.2 Job Success

Imagine you land a sweet tech job, and on your first day your boss says to you, “Implement a recursive algorithm to flatten our data structure.” If your entire prep for this job was reading with very little coding, you might understand each individual word, but actually accomplishing the task would be a disaster.

Now imagine your prep involved using recursion and constructing algorithms over and over again. You would not only understand the words, but also have a strong idea of how to convert those words into working code.

Tip

Be ready for day one on the job. *Do your homework.*

30.1.3 Personal Drive (Grit)

Does the following statement resonate with you? If so, perfect.

“If I stumble, I WILL pick myself up, brush off the dust, and try again.”

All throughout LC101, you will be asked to code. Each exercise, studio and assignment is designed to give you experience through practice. You WILL make mistakes, and that is OK. Often, our mistakes teach us more than getting the correct answer on our first try.

Every genius programmer you see on Facebook or YouTube started out in front of a screen saying, “Oops,” “ARGH!” or “#*&%@#!” No one simply “gets” coding without some trial-and-error.

Tip

Use your mistakes as learning opportunities. *Do your homework.*

30.1.4 Effort = Outcome

Let’s take a look at a sample coding task: “*Prompt the user to enter a number, then print ‘Even’ if it is divisible by 2, otherwise print ‘Odd’.*”

Now let’s take a look at an imaginary student’s attempt at solving this problem:

- | |
|----------------------------|
| 1.
2.
3.
4.
5. |
|----------------------------|

Hmmm. A blank answer space. What might be the reason?

- a. The student did not understand how to solve the problem.
- b. The student knew how to solve the problem and decided to skip the task.
- c. The student tried to solve the problem, could not get the program to work, so deleted the code.
- d. The student ran out of time when trying to complete the prep work before class.

From a teacher’s perspective, ANY of these reasons could be valid, and we have no way of determining which is true. This prevents us from knowing how to best help the student. Where would we begin?

For the student, a blank response provides no benefit because the necessary practice was either ignored or incomplete. Students gain only as much as they put in. SO:

- a. Even if you have no clue how to approach a task, MAKE AN ATTEMPT ANYWAY, then ask questions.
- b. If you know how to solve the problem, COMPLETE THE TASK ANYWAY, because practice makes better. Also, you could use your code to help answer a classmate’s question.

- c. If you tried to solve the problem, but your code did not work, DO NOT DELETE YOUR ATTEMPT. Ask a question. Showing your work to your teacher, TA or classmates will give them a clear idea about your thought process.
 - d. If you ran out of time, GO BACK AND FILL IN THE BLANKS LATER. Practice makes better. If you neglect one set of skills, then the tasks that come later and depend on those skills will be more difficult.
-

Tip

Learning takes work, and you need the practice. *Do your homework.*

30.1.5 Sports Motivational Stuff

Attention sports fans! Embrace your favorite quote(s):

Sport	Quote
Baseball	“There may be people who have more talent than you, but there’s no excuse for anyone to work harder than you do.” - <i>Derek Jeter</i>
Gymnastics	“I’d rather regret the risks that didn’t work out than the chances I didn’t take at all.” - <i>Simone Biles</i>
Football	“I was always willing to work. I was not the fastest or biggest player, but I was determined to be the best football player I could be on the football field, and I think I was able to accomplish that through hard work.” - <i>Jerry Rice</i>
Soccer	“The backbone of success is... hard work, determination, good planning, and perseverance.” - <i>Mia Hamm</i>
Tennis	“If I don’t get it right, I don’t stop until I do.” - <i>Serena Williams</i>
Rocky	How can you listen to this and NOT be inspired? (Gonna Fly now)
Optional	Imagine your favorite motivational phrase here. - <i>Some admired person</i>

Tip

Your heroes worked really hard, so should you. *Do your homework.*

30.1.6 Social Media

Want motivation in 140 characters or less? Try these (LOL):

- a. #hardworkworks
 - b. Rocky ([Gonna Fly Now](#)) because it's just that good.
-

Tip

Do the HW.

30.1.7 Marathon Analogy

Pretend you are not a runner (complete with the “0.0” sticker on your car), but you decide to compete in a marathon. You cannot just drive to the starting line, put on your running shoes and go.

You have to train:

- a. Begin by getting good at running 1 mile.
- b. Then get good at running 3 miles.
- c. Then get good at running 6, then 8, then 10 miles. By now you could try a half-marathon, and proudly slap a “13.1” sticker on your car.
- d. Continue training and increasing your distance. You WILL earn that “26.2” sticker, which will look GREAT when placed in line with 0.0 and 13.1.
- e. Your stickers demonstrate your commitment and might even inspire other non-runners. They will see how you started “just like them” and notice how your effort spurred personal growth.

Learning to code follows the same idea:

- a. Begin with “Hello world!”
- b. Then learn variables, strings and arrays.
- c. Then learn if/else statements and loops, followed by functions and modules.
- d. Then code your first “half-marathon”.
- e. Continue practicing to increase your skills. You WILL earn that marathon.js sticker as you build solid demo projects and complete more interviews.
- f. Welcome, fellow coder. Don’t forget to inspire others.

Tip

Do your homework, and you will consistently get better.

30.1.8 Best Practices

Whew! You made it to the bottom of the page. Good job. Here are some final bits of advice:

1. DO try every exercise, studio and practice problem.
2. Repeated practice helps master the basic syntax quirks for a given programming language.
3. DO experiment. Once your code correctly solves the given task, feel free to tweak it. Great fun can be had if you ask, “What if I try ____,” and then go and do just that. For example, if a problem asks you to sort a list alphabetically, can you order it from z to a instead?
4. ASK FOR HELP when you get stuck. We’ve all been there, and there is no shame in seeking advice. Use your instructors, TAs, classmates, Stack Overflow, and Google as the brilliant resources they are.
5. The only “dumb questions” are the ones that are not asked.
6. The rubber duck method works. Sometimes just describing a coding problem out loud (to your screen, a co-worker, the wall, or a rubber duck) sparks an idea about how to solve it.
7. DO NOT copy/paste answers. There are plenty of websites where you can find complete code posted. A simple copy/paste into the assignment box will give you a correct result, but you have completely skipped your learning opportunity.

And don’t forget:

DO THE HOMEWORK!!!!

CHAPTER
THIRTYONE

INDEX

CHAPTER
THIRTYTWO

GLOSSARY

CHAPTER
THIRTYTHREE

STUDIOS

CHAPTER
THIRTYFOUR

ASSIGNMENTS

34.1 Assignment #1: Candidate Testing

OK, staff, we received many applications for our astronaut training program, and we need to do an initial evaluation of the candidates. Management needs you to create a quick quiz to help select the best candidates.

34.1.1 Requirements

1. Ask the candidate to enter their name
2. Use a loop to ask five questions, one at a time, to the candidate
3. Collect the candidate's answers
4. Check those answers for accuracy (case insensitive equality check)
5. Calculate the candidate's overall percentage
6. Determine if the candidate did well enough to enter our program (need $\geq 80\%$ to pass)
7. Display the results. Example output is listed below.

Example Output

The results output should include the candidate's name, the candidate's responses, the correct answers, the final percentage, and if the candidate passed the quiz.

You are expected to match this format.

```
Candidate Name: Can Twin
1) True or false: 5000 meters = 5 kilometers.
Your Answer: false
Correct Answer: true

2)  $(5 + 3)/2 * 10 = ?$ 
Your Answer: 45
Correct Answer: 40

3) Given the array [8, "Orbit", "Trajectory", 45], what entry is at index 2?
Your Answer: trajectory
Correct Answer: trajectory

4) Who was the first American woman in space?
Your Answer: sally ride
```

(continues on next page)

(continued from previous page)

Correct Answer: sally ride

5) What is the minimum crew size for the International Space Station (ISS) ?

Your Answer: 10

Correct Answer: 3

>>> Overall Grade: 40% (2 of 5 responses correct) <<<

>>> Status: FAILED <<<

Note

The output will vary slightly based on the candidate's answers to each question.

34.1.2 Take It Step by Step

When starting any project, it's best to approach it as a series of smaller, testable parts. The goal is to get simple parts working first and then expand the code in a systematic way. The following is NOT the only way to complete this assignment, but it provides a framework for thinking through the project.

Login to repl.it

If you are enrolled in a repl.it classroom for this course, login to that classroom and open the starter code for the *Candidate Testing* assignment. If you are NOT enrolled in a repl.it classroom, fork this starter repl.it.

Part 1: Minimum Viable Quiz

1. Define variables for:
 - a. candidate's name
 - b. a quiz question (pick any question from the table in step 2)
 - c. the correct answer
 - d. the candidate's response
2. Ask for the candidate's name. Before moving to the next step, use `console.log` to verify that your code correctly stores the information.
3. Display the question and prompt the candidate for an answer. As before, use `console.log` to verify that your program correctly stored the answer.
4. Check the candidate's answer to see if it is correct.
5. Provide basic feedback to the student. This should include their name and whether their answer was correct.

Note

If not already done, remove the extra `console.log` statements from steps 2 & 3. Make sure your small app works properly before moving on to part 2.

Part 2: Use Arrays

Now that your small app is working, expand it to deal with multiple questions.

1. Redefine your question and correct answer variables to be arrays.
2. Fill these arrays with the questions and answers listed in the table below.
3. You still need to ask for the candidate's name.
4. Using bracket notation, select one question and use that to prompt the candidate.
5. Compare the candidate's response to the proper entry in the answers array.
6. Replace the basic feedback with a template literal.

Note

Checking for the correct answer should be case insensitive (e.g. "Orbit" is the same as "orbit").

Question	Answer
True or false: 5000 meters = 5 kilometers.	"True"
(5 + 3)/2 * 10 = ?	"40"
Given the array [8, "Orbit", "Trajectory", 45], what entry is at index 2?	"Trajectory"
Who was the first American woman in space?	"Sally Ride"
What is the minimum crew size for the International Space Station (ISS)?	"3"

Part 3: Use Iteration to Ask All Questions

Add one or more loops to your code to ask all the questions in the quiz. Use arrays to collect and check all the candidate's answers. Finally, calculate the candidate's score and print the results.

Helpful hint - To calculate the candidate's percentage, use the equation:

$$(\text{Number of Correct Answers}) / (\text{Number of Questions}) * 100$$

Note that the final report MUST have the format shown in the "Results Output" section.

34.1.3 Sanity Checks

Before submitting your solution, make sure your program:

1. Does NOT consider case when checking answers.
2. Includes at least one loop and one conditional.
3. Uses at least one template literal.
4. Correctly accepts or rejects a candidate based on their percentage.

34.1.4 Submitting Your Work

1. From the address bar at the top of the browser window, copy the URL of the repl.it that contains your solution.

Example

repl.it classroom URL: <https://repl.it/student/submissions/9999999>

2. Go to the Canvas assignment page and click Submit Assignment.
3. Paste the URL into the Website URL input.
4. Click Submit Assignment again.
5. Notify your TA that your assignment is ready to be graded.

34.2 Assignment #2: Scrabble Scorer

To see how far your skills have come, we are going to give you a mission to update our *super* important Scrabble Scorer program. Did you think you were going to work on Mars rover code already?

The current Scrabble Scorer, is a program that will take in a word and return the point value. The program needs to be rewritten for two reasons. First the data structure used to store the letter point values is inefficient. Second the program should have three scoring algorithms and allow the user to pick which algorithm to use. These new features will make the program more efficient and user friendly.

If you are NOT enrolled in a repl.it classroom for this course, fork the starter code at this [repl.it](#).

34.2.1 Requirements

1. Create an `oldScoreKey` object that contains the previous scoring system.
2. Create a `transform` function which will return a `newScoreKey` object.
3. Create an `initialPrompt` function that asks the user which scoring algorithm to use.
4. Create a `scoringAlgorithms` array to hold three scoring objects, which include different scoring functions.
5. Create a `runProgram` function to serve as the starting point for your program.

Example Output

```
Welcome to the Scrabble score calculator. Click 'Stop' to quit.

Which scoring algorithm would you like to use?

0 - Scrabble: The traditional scoring algorithm.
1 - Simple Score: Each letter is worth 1 point.
2 - Bonus Vowels: Vowels are 3 pts, consonants are 1pt
Enter 0,1,2: 0
Using algorithm: Scrabble

Enter a word to be scored: LaunchCode
Score for 'LaunchCode': 18

Enter a word to be scored: Rocket
Score for 'Rocket': 12

Enter a word to be scored:
```

34.2.2 Detailed Requirements

Transform

Currently the Scrabble Grader software uses the data structure below for the traditional Scrabble scoring algorithm. Take a few moments to review how `oldScoreKey` relates a letter to a point value.

```
1 const
2   1 'A'  'E'  'I'  'O'  'U'  'L'  'N'  'R'  'S'  'T'
3   2 'D'  'G'
4   3 'B'  'C'  'M'  'P'
5   4 'F'  'H'  'V'  'W'  'Y'
6   5 'K'
7   8 'J'  'X'
8 10 'Q'  'Z'
```

The object keys of `oldScoreKey` are the Scrabble points, and the values are arrays of letters. All letters in the array have the Scrabble point value equal to the key. For example, '`A`' is worth 1, and '`J`' is worth 8.

With the old format, to find the point value for a letter, the program must iterate over each key in `oldScoreKey` and then check if the letter is inside the array paired with that key. This search within a search is inefficient.

It would be better to create a `newScoreKey` object that has 26 keys, one for each letter. The value of each key will be the Scrabble point value.

Example of new key storage

- `a` is worth 1
- `j` is worth 8
- `k` is worth 10

In `newScoreKey`, the letters themselves are keys, so a single search will identify a point value.

Example

Example of `newScoreKey` object usage.

```
"Scrabble scoring values for"
"letter a: "
"letter j: "
"letter z: "           "z"
```

Console Output

```
Scrabble scoring values for
letter a: 1
letter j: 8
letter z: 10
```

Write a `transform` function that takes the `oldScoreKey` object as a parameter. `transform(oldScoreKey)` will return an object with the letters as keys. The keys in the `newScoreKey` object should be *lowercase* letters.

Hints

1. Recall that `for...in` loops iterate over the keys within an object.
2. To access the letter arrays within `oldScoreKey`, use bracket notation (`oldScoreKey['key']`).
3. To access a particular element within a letter array, add a second set of brackets (`oldScoreKey['key'][index]`), or assign the array to a variable.

Example

```
1      "Letters with score '4':"          '4'
2      "3rd letter within the key '4' array:"      '4'  2
3
4 let           '8'
5      "Letters with score '8':"
6      "2nd letter within the key '8' array:"      1
```

Console Output

```
Lettters with score '4': [ 'F', 'H', 'V', 'W', 'Y' ]
3rd letter within the key '4' array: V

Lettters with score '8': [ 'J', 'X' ]
2nd letter within the key '8' array: X
```

User Prompts

The current Scrabble Scorer only uses one scoring algorithm. For the new version we want to let the user pick between three algorithms. Define an `initialPrompt` function that will introduce the program and then ask the user which scoring algorithm they want to use. See *Example Output* above and the next section for details on available options.

Scoring Algorithms

Create a `scoringAlgorithms` array that contains three scorer objects. Each object should contain three keys: `name`, `description`, and `scoreFunction`.

The `scoreFunction` for each object should be a function that takes in one parameter named `word` and returns a point value based on the logic listed below. The `scoreFunction` functions can be named or anonymous.

Name	Description	Score Function
Scrabble	The traditional scoring algorithm.	A function with a <code>word</code> parameter that returns a score. Uses the <code>newScoreKey</code> object to determine that score.
Simple Score	Each letter is worth 1 point.	A function with a <code>word</code> parameter that returns a score.
Bonus Vowels	Vowels are 3 pts, consonants are 1pt.	A function with <code>word</code> parameter that returns a score.

Example

```
// Scrabble scoring
    "algorithm name: "          0
        "scoreFunction result: "  0
    ↵"
// Simple scoring
    "algorithm name: "          1
        "scoreFunction result: "  1
    ↵"
// Bonus Vowel scoring
    "algorithm name: "          2
        "scoreFunction result: "  2
    ↵"
```

Console Output

```
algorithm name: Scrabble
scoreFunction result: 24
algorithm name: Simple Score
scoreFunction result: 10
algorithm name: Bonus Vowels
scoreFunction result: 16
```

Note

All three scoring algorithms are case *insensitive*, meaning that they should ignore case.

Tie it All Together

Define a `runProgram` function that will:

1. Accept the `scoringAlgorithms` array as an argument.
2. Use `initialPrompt` to pick the algorithm.
3. Prompt the user for a word to score.
4. Use the selected algorithm to determine the score for the word:
 - a. If the user entered 0 or an invalid option, use the Scrabble `scoreFunction`.
 - b. If the user entered 1, use the Simple Score `scoreFunction`.
 - c. If the user entered 2, use the Bonus Vowels `scoreFunction`.
5. Display the score for the word.
6. Repeat steps 3 to 5 until the program is stopped.

34.2.3 Test Words

Here are some words you can use to test your code:

1. `JavaScript` = 24 points using Scrabble, 10 using Simple Score, and 16 using Bonus Vowels.
2. `Scrabble` = 14 points using Scrabble, 8 using Simple Score, and 12 using Bonus Vowels.
3. `Zox` = 19 points using Scrabble, 3 using Simple Score, and 5 using Bonus Vowels.

34.2.4 Bonus Mission

Score words spelled with blank tiles by adding ' ' to the newScoreKey object. The point value for a blank tile is 0 points.

34.2.5 Submitting Your Work

1. From the address bar at the top of the browser window, copy the URL of the repl.it that contains your solution.

Example

repl.it classroom URL: <https://repl.it/student/submissions/9999999>

2. Go to the Graded Assignment #2 page in Canvas and click *Submit Assignment*.
3. Paste the URL into the Website URL input.
4. Click *Submit Assignment* again.
5. Notify your TA that your assignment is ready to be graded.

34.3 Assignment #3: Mars Rover

This task is going to put your unit tests, modules, and exceptions knowledge to use by writing tests and classes for the Mars rover named Curiosity.

34.3.1 Requirements

1. Fork the [Mars rover starter repl.it](#).
2. Write a unit test for each item in the *Test List* shown below.
 - Some tests have been created for you as examples.
3. Write classes and methods for each *required class and method* shown below.
4. Each class should be defined in it's own file and exported and imported using modules.

34.3.2 Test List

Focus on one test at a time. Write the test and *then* the code to make it pass. Only write the minimum amount of code needed to make the test pass. There are some constraints on how you can implement these features. A list of *required classes and methods* is below.

Each numbered item describes a test. You should use these exact phrases as the test description. You will have 11 tests (12, if you do the bonus) at the end of this assignment.

Message Tests

To be written in `spec/message.spec.js`.

1. Throws error if name NOT passed into constructor as first parameter



Fig. 1: Selfie of Curiosity on Mars.

- This test is provided in the starter code. The code to make it pass is also included.
2. Constructor sets name
 3. Contains commands passed into constructor as 2nd argument

Command Tests

To be written in `spec/command.spec.js`.

4. Throws error if type is NOT passed into constructor as first parameter

Rover Tests

To be written in `spec/rover.spec.js`.

5. Constructor sets position and default values for mode and generatorWatts
6. Response returned by `receiveMessage` contains name of message
7. Response returned by `receiveMessage` includes two results, if two commands are sent in message
8. Responds correctly to status check
9. Responds with correct status after `MODE_CHANGE`
10. Responds with false completed value, if attempt to move while in `LOW_POWER` mode
11. Responds with position for move command

34.3.3 Required Classes and Methods

Message Class

- `constructor(name, commands)`
 - `name` is a string that is the name of the message.
 - `commands` is an array of `Command` objects.

Example

```
let      new      'MODE_CHANGE'  'LOW_POWER'  new      'STATUS_CHECK'  
let      new      'e1'
```

Command Class

- `constructor(commandType, value)`
 - `commandType` is a string that represents the type of command (see *Command Types table* for possible values)
 - `value` is a value related to the type of command.

Example

- 'MODE_CHANGE' is passed in as the commandType
- 'LOW_POWER' is passed in as the value. For a list of all modes, see *Rover Modes table*.

```
let      new      'MODE_CHANGE'  'LOW_POWER'
```

Rover Class

- constructor (position)
 - position is a number representing the rover's position.
 - Sets this.position to position
 - Sets this.mode to 'NORMAL'
 - Sets default value for generatorWatts to 110
- receiveMessage (message)
 - message is a Message object
 - Returns an object containing a result for each Command in message.commands
 - * Specific details about what is returned are in the *Test List*

Example

```
let      new      'MODE_CHANGE'  'LOW_POWER'  new      'STATUS_CHECK'
let      new      'e1'
let      new      98382
let
```

34.3.4 Rover Commands Types

Com-mand	Value sent with command	Result
MOVE	Number representing the position the rover should move to.	{completed: true, position: 88929237}
STA-TUS_CHEK	No values sent with command.	{completed: true, mode: 'NORMAL', generatorWatts: 110, position: 87382098} Values for mode, generatorWatts, position will depend on current state of rover.
MODE_CHANGE	String representing rover mode (see modes)	{completed: true}

Note

The response value for completed will be false if the command could NOT be completed.

34.3.5 Rover Modes

Mode	Restrictions
LOW_POWER	Can't be moved in this state.
NORMAL	None

34.3.6 Bonus Mission

Add the following test that checks for unknown commands in `spec/rover.spec.js`.

12. Responds with completed false and a message for an unknown command

34.3.7 Submitting Your Work

In Canvas, open the Mars Rover assignment and click the “Submit” button. An input box will appear.

Copy the URL for your repl.it project and paste it into the box, then click “Submit” again.

34.4 Assignment #4: HTML Me Something

You’ve learned a bit of HTML and some CSS, but you have likely only used it in bits and pieces so far, adding or modifying content in exercises or pre-existing files. Here, you are going to take another step forward by building an entire page from scratch. You will also get some practice using Git.

There are two parts to this exercise, one focused on HTML and another focused on CSS. HTML makes up the *structure and content* of web pages, while CSS dictates the *visual style*.

Best practices dictate that *content* and *style* should be kept as separate as possible. To that end, we will build the HTML portion of our page first, and afterwards we will add a few styles with CSS. We do this to avoid using HTML tags to change the general appearance of our page. For example, what if we want all of our main headings to be *red*? We can either add this style one time in the CSS file, or we must include `style="color:red"` in EVERY `h1` tag. Especially for large websites, CSS provides the best place to control the overall appearance of a page.

34.4.1 Sections:

1. *Getting Started*
2. *Getting to Work*
3. *Submitting Your Work*

34.4.2 Getting started

Follow the steps below to create a folder for your project and initialize it as a Git repository:

Setup the Project

1. Navigate into the parent folder where you keep all your course materials (e.g. `1c101/` or `code/`). Only you know where that folder lives in your file system, but you want to do something like:

```
$ cd ~/lc101/
```

2. Make a new folder for this assignment:

```
$ mkdir html-me-something
```

Your directory structure should now look like the below (or something similar):

```
lc101/
  |
  +-- html-me-something/
  |
  ... etc
```

3. Within your new `html-me-something/` directory, use the `touch` command to create and save a new file called `index.html`:

```
$ touch index.html
```

Note

The filename `index.html` is a standard convention for the name of the root page of a website. Most web servers will treat `index.html` as the default file to load from a given directory.

4. Open up your new file in a text editor. Add a single line with the following HTML:

```
<p>YOUR NAME</p>
```

5. Save your file.

6. Finally, open up the file in a web browser. You can do this by selecting *File > Open File* in your web browser and navigating to the location of your new HTML file.

- a. If you need help finding where `index.html` is located, use the command `pwd` for “print working directory”:

```
$ pwd
/FolderName/OtherFolderName/lc101/html-me-something
```

- b. The output shows the path to your current directory. Follow this path after selecting *File > Open File* to locate and open `index.html`.
- c. Once you open your file, you should see a blank white page with your name in the top-left corner.

Use Git

Now let's incorporate Git into the picture.

1. Initialize the project as a Git repository.

In your terminal, make sure you are inside your `html-me-something` folder, and then use the `git init` command to initialize that folder as a Git repository:

```
$ pwd
/Users/adalovelace/lc101/html-me-something
```

(continues on next page)

(continued from previous page)

```
$ git init
Initialized empty Git repository in /Users/adalovelace/lc101/html-me-something/.
↳git/
```

Note

Your name is (probably) something other than **adalovelace**.

Now your project is a Git repository, which will enable you to use all of the magic Git powers:

- a. Version-control to manage your changes.
 - b. Syncing your local repository with a remote repository on Github.com.
-

Note

You only have to do the `git init` step once, at the beginning of a project.

2. Check your status.

Back in the terminal, use the `git status` command to check the status of your newly created repo:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    index.html

nothing added to commit but untracked files present (use "git add" to track)
```

This message says a lot of things, but for now, the most important point is that `index.html` is currently “untracked”. We need to `add` and then `commit` the file so that Git can help us manage its changes.

3. Add your work to the repo.

Use the `git add` command to track your `index.html` file so that it will be staged for your next commit:

```
$ git add index.html
```

Now check your status again.

You should see that your change (the creation of the new file) is staged to be committed:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index.html
```

4. You are now ready to `commit` the changes you staged, along with an appropriate message describing what you changed:

```
$ git commit -m "Created index.html file"
```

Check your status again. Your status should be *clean*:

```
$ git status  
On branch master  
nothing to commit, working directory clean
```

Congrats! You are officially up and running with a version-controlled project.

34.4.3 Getting to Work

It's time to build out your page! Dive into each of the two parts below:

1. *Part 1: HTML*
2. *Part 2: CSS*

34.4.4 Submitting your work

When you are ready to submit, complete the following steps:

Github

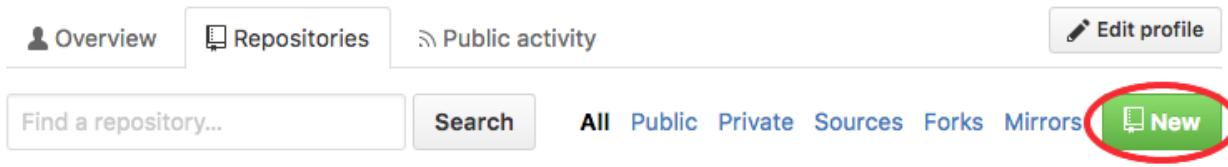
Github.com is a website that hosts Git repositories “in the cloud”. A repository on Github often functions as the central hub for a project, so a developer can do work across multiple machines, or multiple developers can work together on the same project.

For the remainder of this course, you will use Github to submit your work. Here’s how:

1. Create a repo on Github.

In a browser, visit [Github’s website](#). Make sure you are logged into your account (or create an account if you do not already have one).

On your profile page, create a new repository by clicking the green `New` button on the right side of the screen:



Give your repository the same name as your folder, `html-me-something`, and toggle the rest of the options as specified here:

Note

Instead of `LaunchCodeEducation`, you will see your own username.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner



Repository name

 A text input field containing the repository name "html-me-something". To the right of the input is a green checkmark icon.

Great repository names are short and memorable. Need inspiration? How about [jubilant-octo-journey](#).

Description (optional)

 An empty text input field for a repository description.

 Public

Anyone can see this repository. You choose who can commit.

 Private

You choose who can see and commit to this repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

Add a license: **None** ▾



Create repository

2. Pair your local repo with your remote repo.

Now you have two repositories: the local one on your computer, and the remote one on Github. You need to sync them.

The first step to syncing is to give your local repo a *reference* to the remote repo. Using the `git remote` command, you can inform your local repo about the existence of the remote one.

- Copy the url for your remote Github repo as follows:

A screenshot of the GitHub 'Quick setup' interface. It shows a URL: `https://github.com/jesseilev/html-me-something.git`. A yellow arrow points to a copy icon (a clipboard with a plus sign) which is enclosed in a yellow circle.

GitHub Clone Url

- Use the command below, but replace “`PASTE_REPO_URL_HERE`“ with the actual url you copied from part a:

```
$ git remote add origin PASTE_REPO_URL_HERE
```

Note

Unless you've set up an SSH key on your computer and added it to your GitHub account, you should always select the HTTPS version of a repository URL.

By running the `git remote add ...` command on the terminal, you are basically saying:

“Hey local repo. Please meet your new friend, `origin`, a remote repo, whose url is `https://github.com/...`”

Note that the name “`origin`” is simply a standard naming convention for the main remote repo paired with a local repo.

3. Push your local changes up to the remote.

Your local repo is currently *ahead of* your remote repo by a few commits. Locally, you have added and edited a few files, and committed all those changes. However, your remote repo is still entirely empty.

Use the `git push` command to send all your local changes up to the remote:

```
$ git push origin master
```

This command means:

“Hey Git, please push all my local changes to the remote repo called `origin` (specifically, to its `master` branch).”

Refresh the browser window on your Github page, and notice that your HTML and CSS files have appeared!

34.4.5 Turning In Your Work

In Canvas, open the HTML Me Something assignment and click the “Submit” button. An input box will appear.

Copy the URL for your GitHub repo and paste it into the box, then click “Submit” again.

34.4.6 Bonus Mission

If you want to show off your hard work to all your friends, Github has a cool feature called *Github Pages* that makes this really easy.

Github provides free hosting for any “static” web content (like this project). All you have to do is change a setting on your GitHub repository.

1. In a browser, go to the Github page for your repository.
2. Click on the *Settings* tab
3. Scroll down to the *GitHub Pages* section and enable the GitHub Pages feature by choosing your `master` branch from the dropdown. Hit *Save*.

The screenshot shows the GitHub Pages settings interface. At the top, it says "GitHub Pages". Below that, a message states: "GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository." Under the heading "Source", it says "GitHub Pages is currently disabled. Select a source below to enable GitHub Pages for this repository." A dropdown menu is open, showing "master branch ▾" and a "Save" button next to it.

4. In any browser, you should now be able to visit `YOUR_USERNAME.github.io/html-me-something` and see your web page!

APPENDICES

35.1 About This Book

35.1.1 License



Introduction to Professional Web Development in JavaScript by The LaunchCode Foundation is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

35.1.2 Contributors

Chris Bay

Jim Flores

Blake Mills

Sally Steuterman

35.2 Style Guide

When writing code there are specific language rules you must follow for your code to execute. In addition to those language rules there are **style guides** which define style rules to be followed when writing code. These style rules are conventions about how to write the code. For example: how to name variables, where to place brackets, where to put new lines.

Imagine a book written by multiple authors, each of which uses different practices for capitalization, indentation, and punctuation. One writes in all-lowercase and never indents paragraphs. Another capitalizes every word and uses only the period for punctuation. Such a book would be hard to read; reading code can be that bad if everyone isn't following the same style guide conventions.

Benefits of using a Style Guide

- Consistently written code is easier to read
- Predictable variable and file names
- Clear rules to follow when writing code

There are multiple style guides for each language. For the sake of consistency, it's important to follow the style guide rules of your organization or company when writing code. The style guide for this course is below.

35.2.1 JavaScript Style Guide

Camel Case Variable Names

Camel case is defined as starting with a lowercase word and then using uppercase for the first letter of any additional words in the variable name.

Good

```
const      7
const      225
let       false
```

Bad

```
const      7
const      225
let       false
```

Descriptive Variable Names

Variable names should convey meaning.

Good

```
const      7
const      225
let       false
```

Bad

```
const      7
const      225
let       false
```

Opening Braces and Statements on Same Line

Put opening braces on the same line as the statement.

Good

```
if        220
        'WARNING'
else
        'Temp fine'
```

Bad

```
if          220
           'WARNING'

else
           'Temp fine'
```

Always Use Braces for If Statements and Loops

Good

```
if          220
           'WARNING'

for let 0    100
```

Bad

```
if          220
           'WARNING'

for let 0    100
```

Use Semicolons

Good

```
let          200
if          220
           'WARNING'
```

Bad

```
let          200
if          220
           'WARNING'
```

Indent Code Blocks One Tab

Indentation is a key tool for making code readable. Indent one *Tab* inside each **code block**. The definition of what a *Tab* is differs between teams. The important thing is to be consistent and use the same *Tab* throughout your project.

Good

```
const          120 34 15 71 89 94
let           0
for let      0

    "Adding"
    "Total Kilometers"
if           100
    "warning: trip distance longer than advised"

if          1000
    "Over limit for month"
else
    "Still under limit for month"
```

Bad

```
const          120 34 15 71 89 94
let           0
for let      0

    "Adding"
    "Total Kilometers"
if           100
    "warning: trip distance longer than advised"

if          1000
    "Over limit for month"
else
    "Still under limit for month"
```

35.3 Git Workflows

This cheatsheet covers basic workflows for local and remote Git repositories.

For details on the Git commands mentioned, or on creating and cloning repositories, revisit the chapter on *Git More Collaboration*.

35.3.1 Single Developer / Single Branch

When working on a project by yourself, you will typically follow this workflow:

1. Make changes to your code
2. Check status: `git status`
3. Stage changes: `git add .`
4. Commit changes: `git commit -m "Added feature X"`
5. (If working with a remote repository) Push changes: `git push origin master`

35.3.2 Single Developer / Feature Branches

As you become more comfortable with Git and begin to build larger projects on your own, you'll want to use **feature branches**. This is a branch created in order to *isolate* changes related to a *single* feature or bug. Professional developers use them for a few reasons:

- They can work on multiple new features at the same time, while keeping the work separated.
 - If a bug is found, they can commit changes to their feature branch and switch to the branch containing the bug.
 - Working in feature branches allows code to be easily reviewed by other developers before it is merged.
-

Fun Fact

Some development teams will *only* work in feature branches. On these teams, developers are NOT allowed to make changes directly to the `master` branch. They must work in feature branches and create pull requests into `master` to add those changes to the main code base.

In the workflow below, we use `main` to refer to the branch from which the feature branch is created (often `master`) and `feature` to refer to the feature branch. Remember to use descriptive names for your branches.

1. Create a new branch. While in `main` create and move to the feature branch: `git checkout -b feature`.
2. Follow the *Single Developer / Single Branch* workflow until work on the feature is complete.
3. After all changes in `feature` have been committed and pushed, move back to `main`: `git checkout main`.
4. Merge your feature branch back into `main`, resolving any conflicts: `git merge feature`

35.3.3 Team / Single Branch

The case of a team working on a project with a single branch is *very* uncommon. However, multiple developers working on the same branch happens quite a bit. This workflow, therefore, can be thought of as a sub-workflow of the *Team / Feature Branches* workflow.

This workflow is similar to the *Single Developer / Single Branch* workflow, only now you must be mindful to merge in changes made by others.

1. Make changes to your code
2. Check status: `git status`
3. Stage changes: `git add .`
4. Commit changes: `git commit -m "Added feature X"`
5. Pull changes made by others: `git pull origin master`
6. Merge changes as necessary
7. Push changes: `git push origin master`

35.3.4 Team / Feature Branches

The workflow for a team of developers using feature branches combines the *Team / Single Branch* and *Single Developer / Feature Branches* workflows.

In the workflow below, we use `main` to refer to the branch from which the feature branch is created (often `master`) and `feature` to refer to the feature branch. Remember to use descriptive names for your branches.

1. Create a new branch. While in `main` create and move to the feature branch: `git checkout -b feature`. OR if contributing to a branch made by a team member, fetch and checkout their existing branch: `git fetch origin` then `git checkout feature`
2. Follow the *Team / Single Branch* workflow until work on the feature is complete.
3. After all changes in `feature` have been committed and pushed, move back to `main`: `git checkout main`.
4. Merge your feature branch back into `main`, resolving any conflicts: `git merge feature`. Alternatively, create a pull request <create-pr> into `main` as described below.

35.3.5 Working With Pull Requests

The ability to create pull requests is a powerful feature of GitHub that allows changes to be reviewed and discussed by team members.

A **pull request** is a request via GitHub to merge one branch into another. Team members can comment on and review the changes in the request, suggesting or requiring changes. Once the code is ready, the pull request is merged and closed. The code from the feature branch is now part of the destination branch.

Many teams use pull requests when using the *Team / Feature Branches* workflow.

To create a pull request, commit and push all changes in your feature branch. Then visit the project's GitHub page and click on the *Branches* link.

The screenshot shows the GitHub interface for the repository 'chrisbay/communication-log'. At the top, there are navigation links for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Security, Insights, and Settings. Below that, a message says 'No description, website, or topics provided.' There is a 'Manage topics' link. Under the repository name, it shows 9 commits, 2 branches (highlighted with a red oval), 0 releases, and 1 contributor. At the bottom, there are buttons for 'Branch: master ▾', 'New pull request', 'Create new file', 'Upload files', 'Find File', and 'Clone'.

The *Branches* page shows all branches that have been pushed to GitHub. To the right of every branch (except `master`) is a button to create a new pull request.

To create a new pull request, fill out the brief form describing the changes that it contains.

Once the pull request has been created, it remains in the *Open* state for team members to comment.

When the code is ready, the pull request is merged and closed. The code is then part of the destination branch.

35.3.6 Forking a Repository at GitHub

A scenario that will occur from time-to-time in LaunchCode courses, and which occurs quite a lot for developers in general, is when you want to copy another developer's project and modify it. This process is known as "forking a repository" since if you view a project's history as a timeline, copying it effectively creates a "fork" in that history.

Overview Yours Active Stale All branches

Search branches...

Default branch

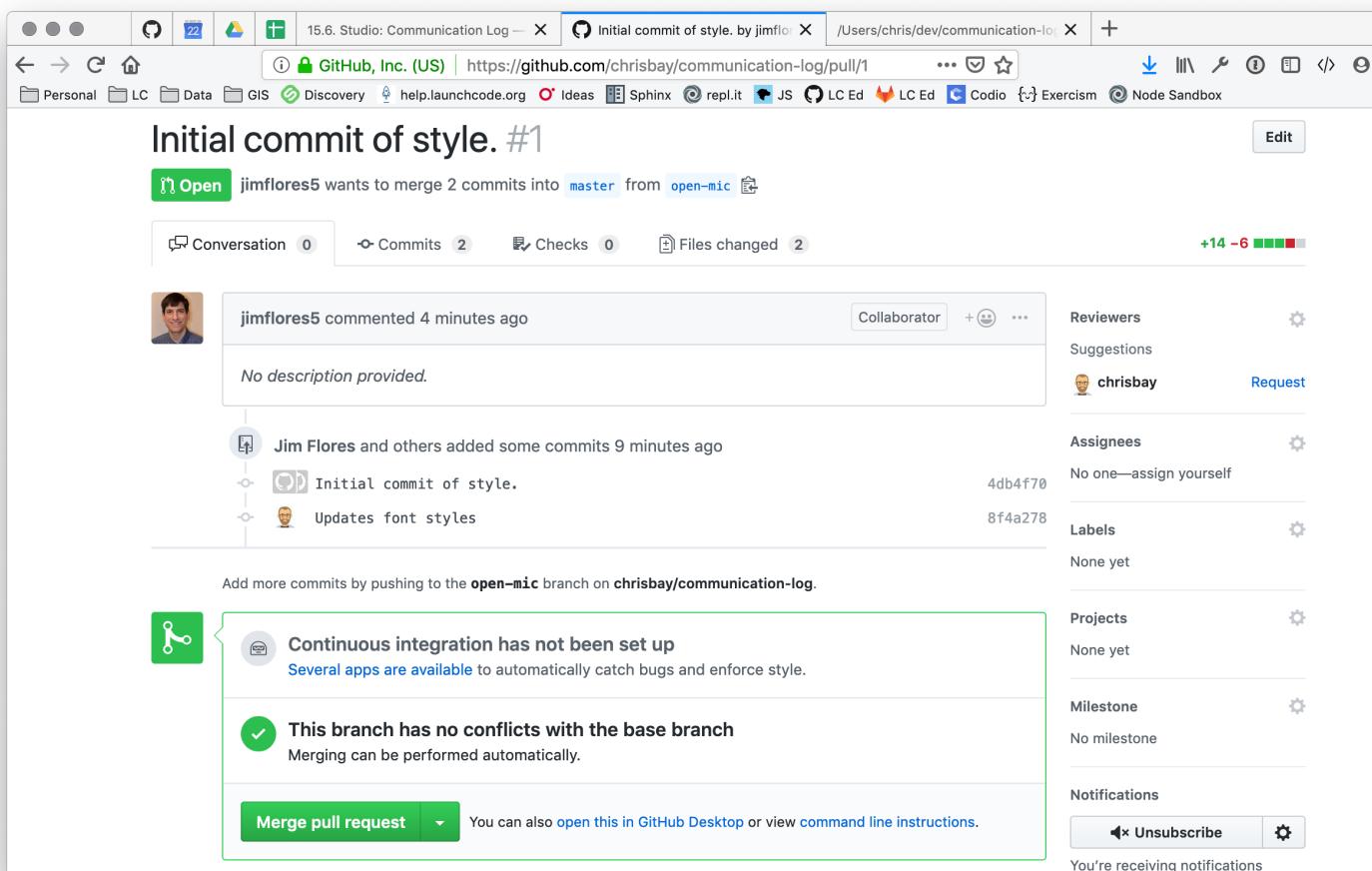
master Updated 3 minutes ago by chrisbay **Default**

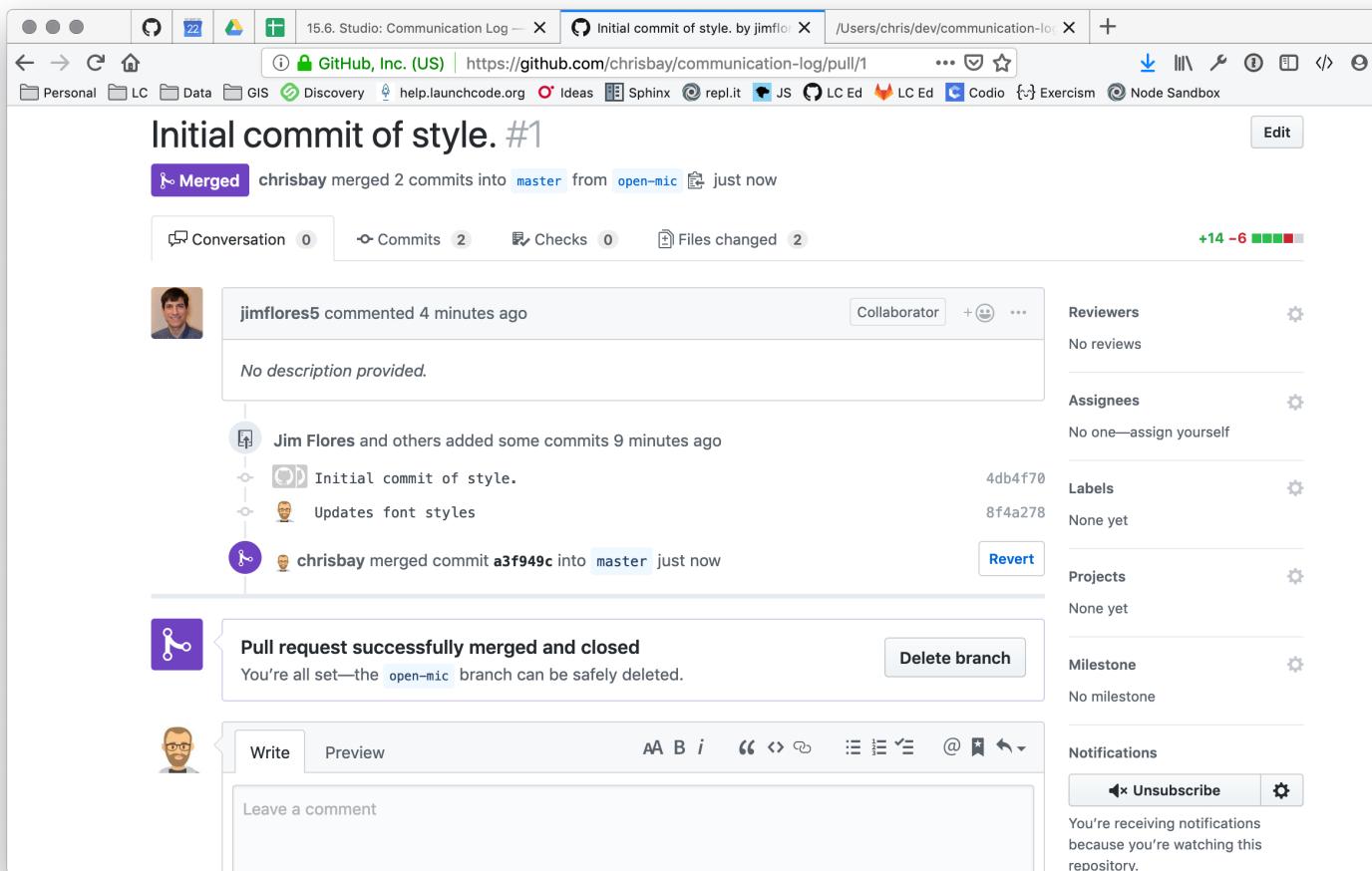
Your branches

open-mic Updated 2 minutes ago by Jim Flores 1 | 1 **New pull req**

Active branches

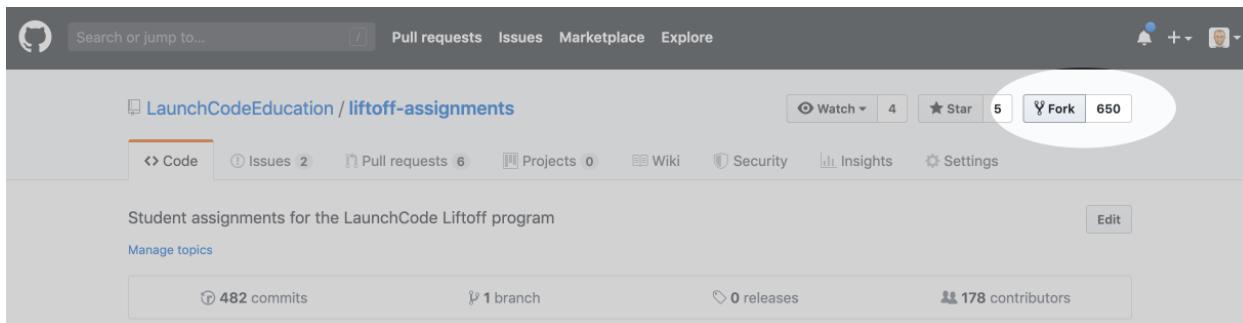
open-mic Updated 2 minutes ago by Jim Flores 1 | 1 **New pull req**





Introduction to Professional Web Development in JavaScript

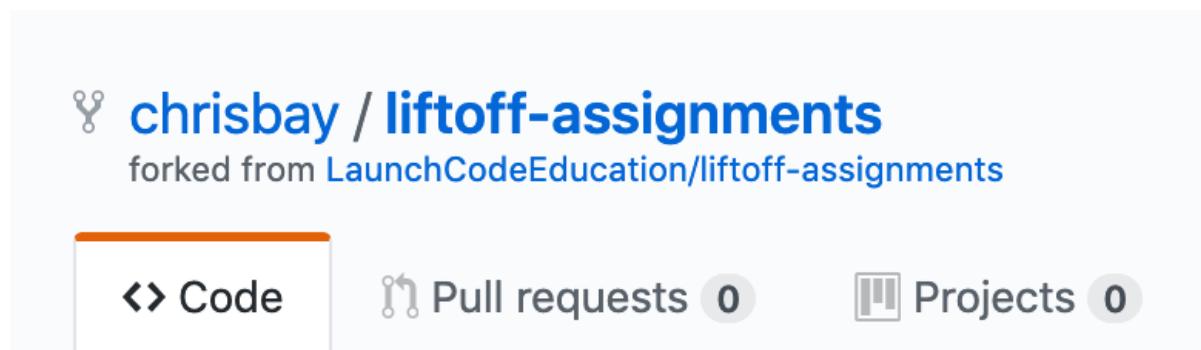
To fork another developer's repository, visit the project at GitHub and hit the *Fork* button:



This will create a *copy* of the remote repository under *your* GitHub profile. You will have a snapshot of the other developer's repository, taken at the moment you hit the *Fork* button.

From your own profile page, you will see the forked repository listed alongside your other repositories. To work on the code, clone the repository to your computer using the method above.

Forked repositories can easily be identified by the reference to the original project under the project name on your profile.



35.3.7 Helpful Git Resources

- [Pro Git Book](#) - A reference book covering Git in depth.
- [Flight Rules for Git](#) - A "How to" guide for git
- [Interactive GitHub Sandbox](#) - A place to practice git without fear of messing anything up.
- [Connecting to GitHub with SSH](#)

35.4 Git Stash

When working with branches in Git, you will sometimes make some changes to your code only to realize you are not working in the branch you thought you were. Thankfully, this is easy to remedy as long as you haven't committed the changes.

This tutorial introduces the `stash` command of `git`, which allows you to easily move the changes from one branch to another

35.4.1 The Situation

We address the following situation:

- You have multiple branches in your local repository. For this tutorial, we'll work with `master` and `feature` branches.
- You are working in a given branch, and have saved some changes.
- Your changes have NOT been staged or committed.
- You want to move your changes to another branch.

If this situation describes you, you're in luck! Let's fix it.

35.4.2 Using Git Stash

Suppose you have a branch called `feature` that you want to work in. You've made some changes, and saved them, only to realize that you're in (Gasp!) the `master` branch.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   main.js

no changes added to commit (use "git add" and/or "git commit -a")
```

At this point, you could try to `git checkout feature`, but you would encounter this error:

```
$ git checkout feature
error: Your local changes to the following files would be overwritten by checkout:
  main.js
Please commit your changes or stash them before you switch branches.
Aborting
```

This error results from the situation in which your `feature` branch has commits that your `master` branch doesn't, so Git can't move the un-staged changes you made in `master` cleanly over to `feature`.

Take a deep breath. This is an easy one to remedy.

Use `git stash` to put these changes off to the side for a moment.

```
$ git stash
Saved working directory and index state WIP on master: 1da4892 Introduce render_
→template
HEAD is now at 1da4892 Introduce render_template
```

Your message will differ, based on the most recent commit that you made in the given branch.

Note that your `master` branch is now “clean”.

```
$ git status
On branch master
nothing to commit, working tree clean
```

Now, safely switch to the `feature` branch.

```
$ git checkout feature
Switched to branch 'feature'
```

And then pick up the changes that you stashed, and put them in the feature branch using `git stash pop`.

```
$ git stash pop
Auto-merging main.js
On branch feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   main.js

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (cb556d3734ed8675b6b81f5d4e37d003bd1bc6b9)
```

That's it! Carry on coding.

35.4.3 References

- [Git Stash Tutorial](#)

35.5 Array Method Examples

35.5.1 concat Examples

The general syntax for this method is:

```
arrayName.concat otherArray1, otherArray2, ...
```

This method adds the elements of one array to the end of another. The new array must be stored in a variable or printed to the screen, because `concat` does NOT alter the original arrays.

Example

```
1      1  2  3
2      'M'  'F'  'E'
3
4
5
6
7
8
9
10
11
12
13
```

Output

```
[1, 2, 3, 'M', 'F', 'E']
[ 'M', 'F', 'E', 1, 2, 3 ]
[ 1, 2, 3, 'M', 'F', 'E', 1, 2, 3 ]
[1, 2, 3]
```

35.5.2 includes Examples

The general syntax for this method is:

```
This method checks if an array contains the item specified in the parentheses (), and returns true or false.
```

Example

```
1 let      1 7 5 9 5
2 let      'hello' 'world!'
3
4           5
5
6           'hi'
```

Output

35.5.3 indexOf Examples

The general syntax for this method is:

```
This method returns the index of the FIRST occurrence of an item in the array. If the item is not in the array, -1 is returned.
```

Example

```
1 let      1 7 5 9 5
2 let      'hello' 'world!'
3
4           5
5
6           'hi'
```

Output

```
2
-1
```

35.5.4 join Examples

The general syntax for this method is:

```
'connecter'
```

join combines all the elements of an array into a string. The *connector* determines the string that “glues” the array elements together.

Example

```
1 let      1 2 3 4
2 let      'hello' 'world' '!'
3 let      ''
4
5         "+"
6
7         ""
8
9
10
11        "_"
12
```

Output

```
1+2+3+4
helloworld!
hello_world_!
```

35.5.5 push And pop Examples

push

The general syntax for this method is:

```
This method adds one or more items to the END of an array and returns the new length.
```

The new items may be of any data type.

Example

```
1 let      'a'  'b'  'c'
2
3         'd'  'f'  42
4
5
```

Output

```
6
['a', 'b', 'c', 'd', 'f', 42]
```

pop

The general syntax for this method is:

This method removes and returns the LAST element in an array.

No arguments are placed inside the parentheses () .

Example

```
1 let      'a'  'b'  'c'  'd'  'e'  
2  
3  
4  
5
```

Output

```
e  
[ 'a', 'b', 'c', 'd' ]
```

35.5.6 reverse Example

The general syntax for this method is:

This method is straightforward - it reverses the order of the elements in the array.

No arguments are placed inside the parentheses () .

Example

```
1 let      'At'  'banana'  'orange'  'apple'  'zoo'  
2  
3  
4
```

Output

```
[ 'zoo', 'apple', 'orange', 'banana', 'At' ]
```

35.5.7 shift And unshift Examples

shift

The general syntax for this method is:

This method removes and returns the FIRST element in an array.

No arguments are placed inside the parentheses () .

Example

```
1 let      'a'  'b'  'c'  'd'  'e'  
2  
3  
4  
5
```

Output

```
'a'  
['b', 'c', 'd', 'e']
```

unshift

The general syntax for this method is:

This method adds one or more items to the START of an array and returns the new length.

The new items may be of any data type.

Example

```
1 let      'a'  'b'  'c'  
2  
3      'hello'  121  
4  
5
```

Output

```
5  
['hello', 121, 'a', 'b', 'c']
```

35.5.8 slice Examples

The general syntax for this method is:

This method copies a range of elements from one array into a new array. `slice` does NOT change the original array, but returns a new array.

The ending index is optional. If it is left out, `slice` returns a new array that includes everything from the starting index to the end of the original array.

If both indices are used, the new array contains everything from the starting index up to, but NOT including the ending index.

Example

```
1 let      'a'  'b'  'c'  'd'  'e'  
2  
3      2  
4  
5      1 4  
6  
7
```

Output

```
[ 'c', 'd', 'e' ]  
[ 'b', 'c', 'd' ]  
[ 'a', 'b', 'c', 'd', 'e' ]
```

35.5.9 `sort` Examples

The general syntax for this method is:

```
[ ]
```

This method arranges the elements of an array into increasing order. For strings, this means alphabetical order. **HOWEVER**, the results are not always what we expect.

Example

Alphabetical order?

```
1 let      'f'  'c'  'B'  'X'  'a'  
2  
3  
4
```

Output

```
[ 'B', 'X', 'a', 'c', 'f' ]
```

From the alphabet song, we know that ‘a’ comes before ‘B’ (and certainly before ‘X’), but JavaScript treats capital and lowercase letters differently. The default sort order places capital letters before lowercase.

Example

```
1 let      'a'  'A'  20  40  
2  
3  
4
```

Output

```
[ 20, 40, 'A', 'a' ]
```

When numbers and strings are sorted, the default order places numbers before all letters.

Example

Numerical sorting.

```
1 let      2   8   10   400   30  
2  
3  
4
```

Output

```
[ 10, 2, 30, 400, 8 ]
```

Here JavaScript gets truly bizarre. How is 8 larger than 400?

When JavaScript sorts, it converts all entries into strings by default. Just like ‘Apple’ comes before ‘Pear’ because ‘A’ comes before ‘P’, the string ‘400’ begins with a ‘4’, which comes before any string starting with an ‘8’. Looking only at the first digit in each number, we see the expected progression (1, 2, 3, 4, 8).

Later in this course, we will explore ways to fix this issue and correctly sort numerical arrays.

35.5.10 `splice` Examples

The general syntax for this method is:

```
of
```

Inside the parentheses (), only the first argument is required.

The `splice` method modifies one or more elements anywhere in the array. Entries can be added, removed or changed. This method requires practice.

Hang on, here we go:

Example

Given only one argument, `splice(index)` removes every entry from `index` to the end of the array.

```
1 let      'a'  'b'  'c'  'd'  'e'  'f'  
2       2      //Everything from index 2 and beyond is removed.  
3  
4
```

Output

```
[ 'a', 'b' ]
```

Example

With two arguments, `splice(index, number of items)` starts at `index` and removes the number of items.

```
1 let      'a'  'b'  'c'  'd'  'e'  'f'  
2  
3     2 3      //Start at index 2 and remove 3 total entries.  
4  
5  
6     1 1      //Start at index 1 and remove 1 entry.  
7
```

Output

```
[ 'a', 'b', 'f' ]  
[ 'a', 'f' ]
```

Example

Given three or more arguments, `splice(index, 0, new item)` starts at `index` and *ADDS* the new items.

```
1 let      'a'  'b'  'c'  'd'  'e'  'f'  
2  
3     2 0 'hello'    //Start at index 2, remove 0 entries, and add 'hello'.  
4
```

Output

```
[ 'a', 'b', 'hello', 'c', 'd', 'e', 'f' ]
```

Example

Given three or more arguments, `splice(index, number of items, new items)` starts at `index` and *REPLACES* the number of items with the new ones.

```
1 let      'a'  'b'  'c'  'd'  'e'  'f'  
2  
3     2 3 'hello'  9  
4     ↵//Start at index 2, replace 3 entries with 'hello' and 9.
```

Output

```
[ 'a', 'b', 'hello', 9, 'f' ]
```

35.5.11 `split` Examples

The general syntax for this method is:

```
'delimiter'
```

`split` is actually a string method, but it complements the array method `join`.

`split` divides a string into smaller pieces, which are stored in a new array. The **delimiter** argument determines how the string is broken apart.

Example

```
1 let      "1,2,3,4"
2 let      "Rutabaga"
3 let      "Bookkeeper of balloons."
4 let
5
6     ','    //split the string at each comma,
7
8
9     ' '    //split the string at each space,
10
11
12    ''    //split the string at each character.
```

Output

```
['1', '2', '3', '4']
['Bookkeeper', 'of', 'balloons']
['R', 'u', 't', 'a', 'b', 'a', 'g', 'a']
```

35.6 DOM Method Examples

35.6.1 `confirm` Examples

The general syntax for this method is:

```
let      "Message to user"
```

Displays a dialog box with a message and returns `true` if user clicks “ok” and `false` if user clicks “cancel”. The browser waits until the user clicks “ok” or “cancel”.

Example

```
let      "Would you like to play a game?"
// Code does NOT continue until user responds to confirm window
if
    "Let's play a board game"
else
    "Oh well, let's code instead"
```

Note

Remember that methods defined on `window` can be used without referencing the `window` variable.

Example

```
1 <!DOCTYPE html>
2   html
3     head
4       title DOM Examples  title
5     head
6   body
7     h1 Window Confirm Example  h1
8     script
9       let                      "Are you excited?"
10      if
11        "Yay! Me too!"
12      else
13        "Oh no! I hope tomorrow is a better day!"
14
15     script
16   body
17 html
```

Console Output (If “cancel” clicked)

```
Oh no! I hope tomorrow is a better day!
```

35.6.2 `getElementById` Examples

The general syntax for this method is:

```
let          "element-id"
```

Sets the `HTMLDocument` for an element that has an `id` attribute that matches the string parameter. Returns a reference to the matching element object if match found. If NO matching element is found, `null` is returned.

Example

```
1 <!DOCTYPE html>
2   html
3     head
4       title DOM Examples  title
5     head
6   body
7     h1 getElementById Example  h1
8     p id "description"
9       This will be turned blue.
10    p
11    script
12      let          "description"
13        "paragraph contents:"
14        "blue"
15    script
```

(continues on next page)

(continued from previous page)

```
16   body  
17     html
```

Console Output

```
paragraph contents: This will be turned blue.
```

Tip

Because `getElementById` returns `null` if an element with a matching id can NOT be found, you could see a message like `TypeError: paragraph is null`. Be sure to double check the id you are using in JavaScript and in the HTML.

35.6.3 `querySelector` and `querySelectorAll` Examples

`querySelector`

The general syntax for this method is:

```
let           "CSS selector"
```

Uses a CSS selector pattern and CSS selector rules to find a matching element. Returns the FIRST matching element. If NO match is found, `null` is returned.

CSS Selector Examples

- class selector: `document.querySelector(".class-name");`
- tag selector: `document.querySelector("div");`
- id selector: `document.querySelector("#main");`

Tip

You can use any valid CSS selector with `querySelector`. The selectors can be simple like `querySelector("div")` or complex like `querySelector("#main div .summary")`.

Example

```
1  <!DOCTYPE html>  
2  html  
3    head  
4      title DOM Examples  title  
5      style  
6        font-weight bold  
7  
8      style  
9    head  
10   body  
11     h1 querySelector Example  h1
```

(continues on next page)

(continued from previous page)

```

13  p id "description" class "main"
14      querySelector's power is exceeded only by it's mystery.
15  p
16  div id "response"
17      It's not that mysterious, querySelector selects elements
18      using the same rules as CSS selectors.
19  div
20  script
21      // selects the <p> using class selector
22      let                               ".main"
23
24          "blue"
25
26      // Selects the <div> using tag selector
27      let                               "div"
28
29          "red"
30  script
31  body
32  html

```

Console Output

```

querySelector's power is exceeded only by it's mystery.
It's not that mysterious, querySelector selects elements
using the same rules as CSS selectors.

```

querySelectorAll

The general syntax for this method is:

```
let           "CSS selector"
```

Uses a CSS selector pattern and CSS selector rules to find matching elements. Returns ALL elements that match the selector. If NO match is found, `null` is returned.

Example

```

1  <!DOCTYPE html>
2  html
3      head
4          title DOM Examples title
5          style
6
7              color red
8
9
10         color purple
11
12         style
13     head
14     body
15         h1 querySelectorAll Example h1

```

(continues on next page)

(continued from previous page)

```
16      h2 Red Fruits  h2
17      ul class "red"
18          li Strawberry  li
19          li Raspberry  li
20          li Cherry  li
21      ul
22
23
24      h2 Purple Fruits  h2
25      ul class "purple"
26          li Blackberry  li
27          li Plums  li
28          li Grapes  li
29      ul
30
31      script
32          // Selects ALL the <li> elements and adds text to each one
33          let                         "li"
34          for let  0
35              " is yummy"
36
37
38          // Selects the PURPLE <li> elements and make them bold
39          let                         ".purple li"
40          for let  0
41              "!!!!"
42
43
44          // Console log the contents of the first items in each list
45          // Remember that querySelector returns only the FIRST match
46          let                         ".red li"
47              "contents of first red li:"
48          let                         ".purple li"
49              "contents of first purple li:"
50      script
51      body
52      html
```

Console Output

```
contents of first red li: Strawberry is yummy dom-demo.html:45:13
contents of first purple li: Blackberry is yummy!!!
```

35.6.4 innerHTML Examples

The general syntax for the `innerHTML` property is:

The `innerHTML` property of elements *reads* and *updates* the HTML and or text that is inside the element.

Note

The innerHTML value for empty elements is empty string "".

Example

```
1 <!DOCTYPE html>
2   html
3     head
4       title DOM Examples  title
5     head
6   body
7     h1 innnerHTML Example  h1
8
9     h2 Yellow Fruits  h2
10    ul class "yellow"
11      li Banana  li
12    ul
13
14    script
15      let
16        ".yellow"
17
18    script
19  body
  html
```

Console Output

```
<li>Banana</li>
```

Tip

Use `.trim` to remove the whitespace around the value of `.innerHTML`

As mentioned above `innerHTML` can be used to read and *update* the contents of an element. `innerHTML` is so powerful that you can pass in strings of HTML.

Example

```
1 <!DOCTYPE html>
2   html
3     head
4       title DOM Examples  title
5     head
6   body
7     h1 innnerHTML Example  h1
8
9     h2 Yellow Fruits  h2
10    ul class "yellow"
11      li Banana  li
12    ul
13
14    script
15      let
16        ".yellow"
17        // Add a <li> to the list
```

(continues on next page)

(continued from previous page)

```
17          "<li>Lemon</li>"  
18  
19      script  
20  body  
21  html
```

Console Output

```
<li>Banana</li>  
<li>Lemon</li>
```

35.6.5 style property Examples

The general syntax for the `style` property:

```
The style property is an object that allows you to read and update the INLINE style properties of the element. The style property does NOT read or update styles defined in a <style> tag or an external CSS file linked with a <link> tag.
```

Example

```
1  <!DOCTYPE html>  
2  html  
3    head  
4      title DOM Examples  title  
5    head  
6    body  
7      h1 style property Example  h1  
8      div id "strawberry" style "color: red;" Strawberry  div  
9      div id "blackberry" style "color: purple; font-size: 5px" Blackberry  div  
10  
11      script  
12        let                      "#strawberry"  
13  
14        let                      "#blackberry"  
15  
16        // Update the font size of strawberry  
17        "45px"  
18  
19      script  
20    body  
21  html
```

Console Output

```
red  
5px  
45px
```

35.7 String Method Examples

35.7.1 indexOf Examples

The general syntax for this method is:

Given a candidate substring, this method returns the integer index of the *first* occurrence of the substring in the string. If the substring does not occur in the string, -1 is returned.

Example

```
1      "LaunchCode"      "C"
2
3      "LaunchCode"      "A"
4
5      "dogs and dogs and dogs!"      "dog"
```

Output

```
6
-1
0
```

Example

An email address must contain an @ symbol. Checking for the presence of this symbol is a part of email address verification in most programs.

```
1 let      "fake.email@launchcode.org"
2 let      "@"
3
4 if      1
5         "Email contains @"
6 else
7         "Invalid email"
8
```

Output

```
Email contains @
```

35.7.2 toLowerCase Examples

The general syntax for this method is:

This method returns a copy of `stringName` with all uppercase letters replaced by their lowercase counterparts. It leaves non-alphabetic characters unchanged.

```
"LaunchCode"
```

Output

```
launchcode
```

Example

The domain portion of an email address (the portion after the @ symbol) is case-insensitive. Emails with domain `launchcode.org` are the same as those with domain `LAUNCHCODE.ORG`. By convention, the all-lowercase version is typically used by an application.

This program standardizes an email address by converting it to all lowercase characters.

```
let      "fake.email@LAUNCHCODE.ORG"  
let
```

Output

```
fake.email@launchcode.org
```

Warning

This example is a bit crude, since the portion of an email address *before* the @ symbol is allowed to be case-sensitive. When standardizing the case of an email in a real application, we would want to be more precise and only convert the domain portion to lowercase characters.

35.7.3 `toUpperCase` Examples

The general syntax for this method is:

```
stringName.toUpperCase()
```

This method returns a copy of `stringName` with all lowercase letters replaced by their uppercase counterparts. It leaves non-alphabetic characters unchanged.

Example

```
1      "LaunchCode"  
2      "launchcode"  
3      "LaunchCode's LC101"
```

Output

```
LAUNCHCODE  
LAUNCHCODE  
LAUNCHCODE'S LC101
```

35.7.4 `trim` Examples

The general syntax for this method is:

This method returns a copy of the string with any leading or trailing whitespace removed. Whitespace characters are those that do not display anything on the screen, such as spaces and tabs.

Example

```
1     "Saint Louis "
2     " Saint Louis"
3     " Saint Louis "
```

Output

```
Saint Louis
Saint Louis
Saint Louis
```

Example

When typing an email address into a web site, a user may inadvertently type a space before and/or after the email address. We can clean up such input using the `trim` method.

This example cleans up user input with `trim`.

```
let      " fake.email@launchcode.org "
let
```

Output

```
fake.email@launchcode.org
```

35.7.5 `replace` Examples

The general syntax for this method is:

Given a search string `searchChar` and a replacement value `replacementChar`, this method returns a copy of `stringName` with the *first* occurrence of `searchChar` replaced by `replacementChar`.

Note

The `replace` method can be used in more powerful ways utilizing regular expressions. We will not cover those here, but you can [read more at MDN](#).

Example

```
"carrot"      "r"  "t"  
"Launch Code"    " "  ""
```

Output

```
catrot  
LaunchCode
```

Example

Some email providers, including Gmail, allow users to put a . anywhere before the @ symbol. This means that fake.email@launchcode.org is the same as fakeemail@launchcode.org.

Remove the . before the @ symbol in an email address.

```
let      " fake.email@launchcode.org "  
let      ". "  ""
```

Output

```
fakeemail@launchcode.org
```

This example illustrates a common use case of `replace`, which is to *remove* a character by replacing it with the empty string.

Warning

Notice in the last example that if there is not a . before the @ symbol, the . that is part of the domain, launchcode.org would be inadvertently removed. In a real application, we would want to isolate the portion in front of @ using `slice`.

35.7.6 `slice` Examples

The general syntax for this method is:

```
"LaunchCode"  0  6
```

Given a starting index `i` and an optional ending index `j`, return the substring consisting of characters from index `i` through index `j-1`. If the ending index is omitted, the returned substring includes all characters from the starting index through the end of the string.

```
"LaunchCode"  0  6  
"LaunchCode"  6
```

Output

```
Launch  
Code
```

On some websites, the portion of an email address before the @ symbol is used as a username. We can extract this portion of an email address using `slice` in conjunction with `indexOf`.

Example

```
1 let      "fake.email@launchcode.org"
2 let      "@"
3 let      0
4
```

Output

```
fake.email
```

35.8 Math Method Examples

35.8.1 Math.abs Examples

The general syntax for this method is:

```
Math.abs(number)
```

This method returns the positive value of a number, which can be printed or stored in a variable. `abs` works on both integer and decimal values.

Numerical strings can also be evaluated, but should be avoided as a best practice.

Example

```
1 let      3
2
3
4      4.44
5      '-3.33'
6
7      24  3
8 // 24/-3 = -8
```

Console Output

```
3
4.44
3.33
8
```

`Math.abs` also works on arrays, but to make the process work, we must combine it with the *map array method*. The syntax for this is:

```
arrayName.map(Math.abs)
```

Note that `Math.abs` takes no argument when applied to an array.

Example

```
1 let      2 3   4.44  "8.88"  
2  
3 let  
4  
5
```

Console Output

```
[2, 3, 4.44, 8.88]
```

35.8.2 Math.ceil, floor, and trunc Examples

Math.ceil

The general syntax for this method is:

```
Math.ceil(number)
```

This method rounds a decimal value UP to the next integer (hence the *ceiling* reference in the name). Integer values remain the same.

`ceil` also operates on arrays (*see below*).

Numerical strings can also be evaluated, but should be avoided as a best practice.

Example

```
1           8.88  
2  
3           8.1  
4  
5           3.9  
6  
7           5
```

Console Output

```
9  
9  
-3  
5
```

Math.floor

The general syntax for this method is:

```
Math.floor(number)
```

This method is the opposite of `Math.ceil`. It rounds a decimal value DOWN to the previous integer. Integer values remain the same.

`floor` also operates on arrays (*see below*).

Numerical strings can also be evaluated, but should be avoided as a best practice.

Example

```
1          8.88  
2  
3          8.1  
4  
5          3.9  
6  
7          5
```

Console Output

```
8  
8  
-4  
5
```

`Math.trunc`

The general syntax for this method is:

```
Math.trunc(number)
```

This method removes any decimals and returns only the integer part of `number`.

`trunc` also operates on arrays (*see below*).

Numerical strings can also be evaluated, but should be avoided as a best practice.

Example

```
1          8.88  
2  
3          10.000111
```

Console Output

```
8  
10
```

Note

At first glance, `Math.floor` and `Math.trunc` appear to do exactly the same thing. However, a closer look shows that the two methods treat negative numbers differently.

Console Output

Combine with `map`

When combined with the `map array method`, `ceil`, `floor`, and `trunc` will operate on each entry in an array. The syntax for this is:

```
arrayName.map(Math.method)
```

Example

```
1 let          2 3.33  4.44  8.88  
2  
3  
4  
5
```

Console Output

```
[ -2, 4, -4, 9 ]  
[ -2, 3, -5, 8 ]  
[ -2, 3, -4, 8 ]
```

35.8.3 `Math.max` and `Math.min` Examples

`Math.max`

The general syntax for this method is:

```
Math.max(x, y, z, ...)
```

This method finds and returns the largest value from a set of numbers (x, y, z, ...).

To find the maximum value in an array, *see below*.

Example

```
2 3 100.01 0 5.2 100
```

Console Output

```
100.01
```

`Math.min`

The general syntax for this method is:

```
Math.min(x, y, z, ...)
```

This method finds and returns the smallest value from a set of numbers (x, y, z, ...).

To find the minimum value in an array, *see below*.

Example

```
2 3 100.01 0 5.2 100
```

Console Output

```
-5.2
```

Max and Min of an Array

Unfortunately, the `max` and `min` methods will NOT take an array of numbers as an argument. There are numerous workarounds. Here are TWO possible solutions.

Sort First

This approach uses the syntax from *the sorting studio* to first order the array. The maximum (or minimum) value can then be identified with bracket notation.

Examples

```
1 let      2 3.33 4.44 8.88
2
3 let          function      return
4
5   `Min = ${0}, Max = ${1}`
```

Console Output

```
[ -4.44, -2, 3.33, 8.88 ]
Min = -4.44, Max = 8.88
```

Alternatively, we could put the array in decreasing order:

```
1 let      2 3.33 4.44 8.88
2
3 let          function      return
4
5   `Max = ${0}, Min = ${1}`
```

Console Output

```
[ 8.88, 3.33, -2, -4.44 ]
Max = 8.88, Min = -4.44
```

Using Spread Syntax

An alternative to the sorting approach described above is to use the **spread operator** (`...`), also called *spread syntax*.

In cases where a set of numbers or strings (x, y, z, etc.) is expected, an array cannot be used as-is. The spread operator expands an array into a comma-separated set of elements, which can be passed as arguments in a function call. `functionName(...[x, y, z])` is identical to `functionName(x, y, z)`.

Example

```
1 let      2 3 100.01 0  5.2 100
2
3 let
4 let
5
6   `Min = ${    }, Max = ${    }`
```

Console Output

```
2 3 100.01 0 -5.2 100
Min = -5.2, Max = 100.01
```

Note the absence of brackets, `[]`, around the numbers printed by line 6. `console.log(...numbers)` executes as `console.log(2, 3, 100.01, 0, -5.2, 100)`, so the output is NOT an array.

Note

The sorting approach works in all browsers. The spread operator, while very convenient, is NOT compatible with Internet Explorer or older versions of other browsers (pre-2015). For more details on the spread operator and its compatibility, check the [MDN Web Docs](#).

35.8.4 Math.pow and Math.sqrt Examples

Math.pow

The general syntax for this method is:

```
Math.pow(x, y)
```

This method calculates and returns the value of x raised to the power of y (x^y), and it is identical to the `x**y` operation. The *pow* name refers to the operation *to the power of*.

Example

```
1   3 4
2     3 4
3 // 3 raised to the power of 4 = 3*3*3*3
```

Console Output

```
81
81
```

Math.sqrt

The general syntax for this method is:

```
Math.sqrt(number)
```

This method calculates and returns the square root of `number`, and it is a shortcut for using the fraction `1/2` in the `pow` method.

Numerical strings can also be evaluated, but should be avoided as a best practice.

Example

```
1          81
2          81 1 2
3
4          111
5          "36"
```

Console Output

```
9
9
10.535653752852738
6
```

`Math.sqrt` also works on arrays, but must be combined with the *map array method*. The syntax for this is:

```
arrayName.map(Math.sqrt)
```

Example

```
1 let      2 16 100 121
2
3
```

Console Output

```
[ 1.4142135623730951, 4, 10, 11 ]
```

35.8.5 Math.random Examples

The general syntax for this method is:

```
Math.random()
```

This method returns a random decimal value between 0 and 1, which can be stored in a variable or used in a calculation.

Note that 0 is a possible selection, but 1 is NOT.

Example

```
1 for 0 5
2   let
3
4
```

Console Output

```
0.34992592600591066
0.11861535165960668
0.019710093901842862
0.7751799992655235
0.46782849511194136
```

Generate a Random Integer

If a random integer must be generated, the result of `Math.random()` can be manipulated with operators (+, -, *, /) and other `Math` methods.

The trick to creating a random integer is to multiply `Math.random()` by a whole number and then round the result to remove the decimal portion. The choice of using the `ceil`, `floor`, or `round` method affects the numbers generated.

Explore the example below:

Example

```
1 for 0 5
2   let
3     `floor = ${Math.floor(Math.random() * 10)}, ceil = ${Math.ceil(Math.random() * 10)}, round = ${Math.round(Math.random() * 10)}`
4
```

Console Output

```
floor = 0, ceil = 1, round = 0
floor = 6, ceil = 7, round = 7
floor = 2, ceil = 3, round = 3
floor = 8, ceil = 9, round = 8
floor = 9, ceil = 10, round = 10
```

After multiplying `Math.random()` by 10, applying the `floor` method gives numbers between 0 and 9. Using the `ceil` method shifts the range up one digit, generating values between 1 and 10. Using the `round` method gives the widest range, generating numbers between 0 and 10.

Rather than trying to remember which method to use, one choice is to **ALWAYS** use `floor` to round to an integer:

1. `Math.floor(Math.random() * 10)` generates a number from 0 - 9.
2. `Math.floor(Math.random() * 120)` generates a number from 0 - 119.

To start our range at 1, just add 1 to the rounded value:

1. `Math.floor(Math.random() * 10) + 1` generates a number from 1 - 10.
2. `Math.floor(Math.random() * 120) + 1` generates a number from 1 - 120.

By changing the value that multiplies `Math.random()` we specify the range for the numbers we want to generate.

1. `Math.floor(Math.random() *maxValue)` generates a number from 0 to `(maxValue-1)`.
 2. `Math.floor(Math.random() *maxValue) + 1` generates a number from 1 to `maxValue`.
-

Try It

Explore generating random numbers at this [repl.it](#).

1. Generate random integers from 1 - 100.
 2. Generate random integers from -5 - 0, but NOT including 0.
 3. *Challenge:* Generate random integers from 20 - 30.
-

35.8.6 `Math.round` Examples

The general syntax for this method is:

```
Math.round(number)
```

This method returns `number` rounded to the nearest integer value.

Numerical strings can also be evaluated, but should be avoided as a best practice.

Example

```
1          1.33
2          28.7
3          8.5
4          "101.45"
```

Console Output

```
1
-29
9
101
```

`Math.round` also works on arrays, but must be combined with the *map array method*. The syntax for this is:

```
arrayName.map(Math.round)
```

Example

```
1 let          1.33  4  8.5   15.523  8.49
2
3
```

Console Output

```
[ 1, 4, 9, -16, 8 ]
```

35.9 Terminal Commands

35.9.1 Terminal Commands Tutorial

As mentioned in the terminal chapter, the essence of the command line is text. Since this is quite different from how many may be used to using their computers, this tutorial is meant to help you picture how your machine is responding when you input common commands into the terminal.

Your computer is basically a file storage system. Sure, you may have many applications installed. But where do they all live? In folders. Aka, directories. The basics of terminal usage involve navigating these directories.

Let's take a look at a given project opened in your VSCode editor:

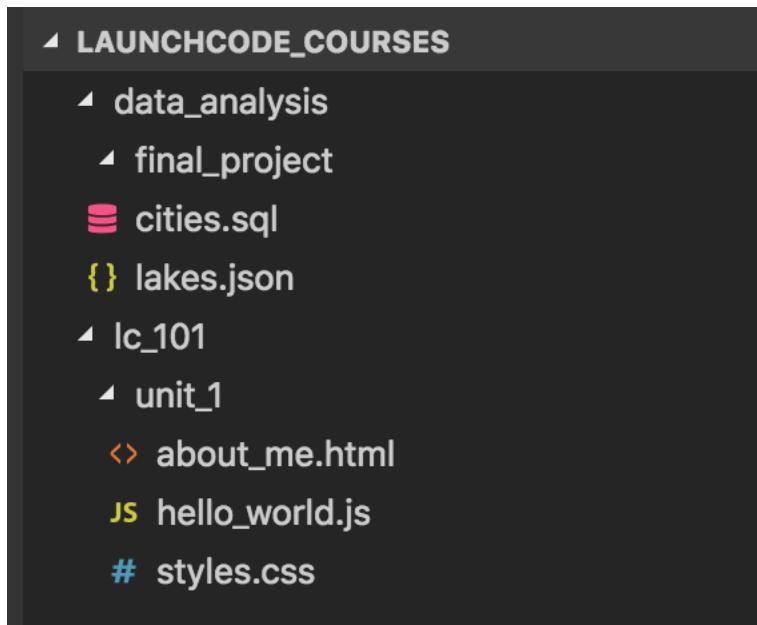


Fig. 1: Sample file tree in VSCode

When working in the terminal, it can be helpful to think of yourself as physically inside of the project's file system. File trees, like the one above, are common visualization tools. Here's another map-like option for imagining your file system:

We'll navigate through and edit this sample project folder for the remainder of this tutorial.

Current Directory (.)

Imagining you are inside of this file system, . is a reference to your location, or **current directory**.

Starting at the top directory, launchcode_courses, . represents your current location.

Here, your terminal will look something like this:

```
computer:launchcode_courses user$
```

Most of what you see to the left of the command prompt symbol, \$ will be different on your machine. The basic structure here is <machine_name>:<current_directory> <user_name>\$.

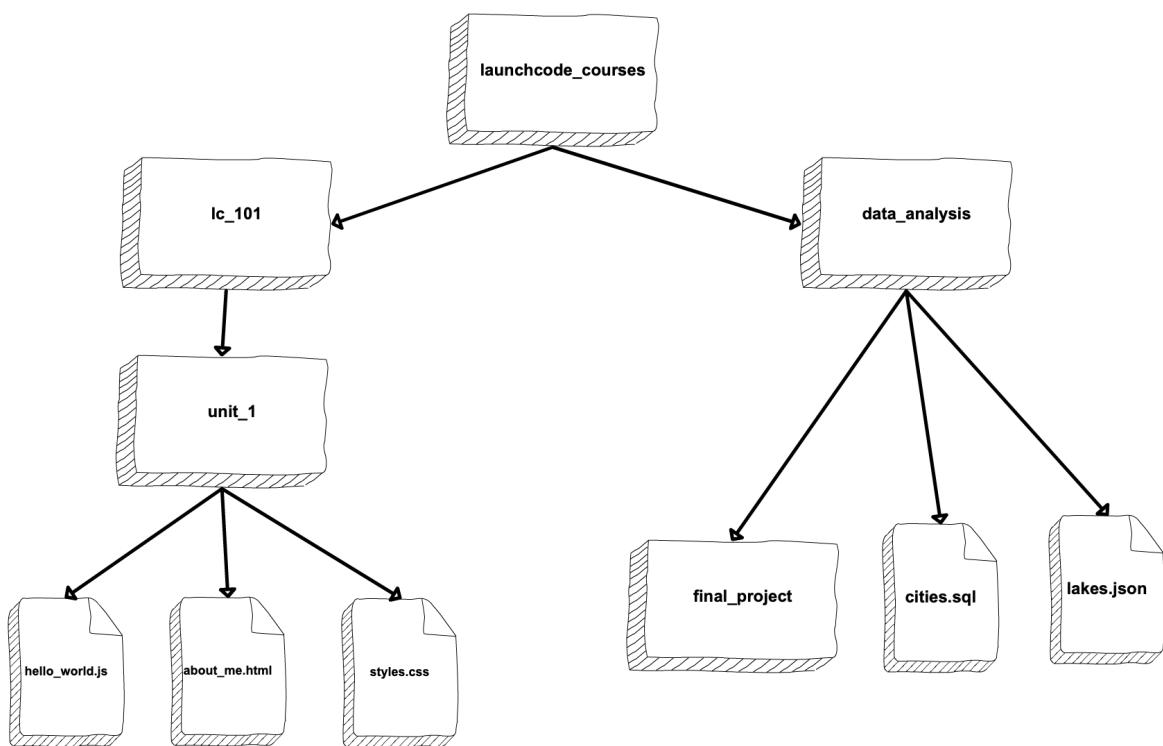


Fig. 2: Sample file system map

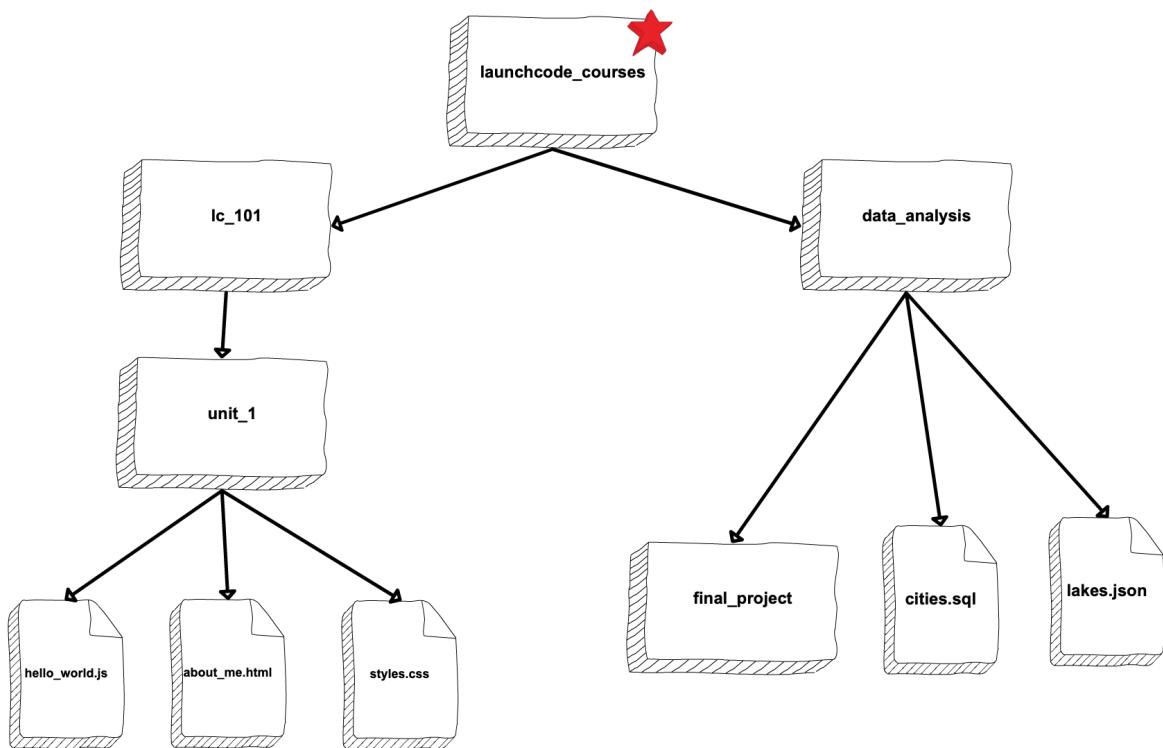


Fig. 3: Your current directory is `launchcode_courses`.

Note

Some users choose to alter what they see before the command prompt. For the purposes of this tutorial, we will simply use <current_directory> \$ as the prompt.

. itself is not a command. If you type only . into the terminal, you're not really telling the machine to do anything just yet.

If you're curious, try it.

Note

Most commands require you to press *Enter* when you are ready to run.

You will probably see a somewhat cryptic message, like this:

```
1 launchcode_courses $ .
2 bash: ..: filename argument required
3 ..: usage: .. filename arguments
4 launchcode_courses $
```

That's ok! Basically, we just entered an incomplete command. Our syntax wasn't quite right. Keep reading and we'll see how to properly use ..

If you move into lc_101, . then refers to that directory. We'll cover how to move locations in detail down in *cd Command*.

```
1 launchcode_courses $      ./lc_101/
2 lc_101 $
```

You may notice that the <current_directory> has updated but apart from that, the computer doesn't give us much response. This is quite common and is a reason why our file system visuals come in handy to help remind us what we're doing.

Back in our map, we've done this:

Parent Directory (..)

.. is a reference to your **parent directory**, aka the directory that CONTAINS your current location.

Remember the VSCode file tree? That containment structure is represented through indentation:

By the end of the *Current Directory* (.), we found ourselves inside of lc_101.

launchcode_courses is the parent directory of both the lc_101 and data_analysis directories. While we're in lc_101, .. refers to launchcode_courses.

Moving further down into unit_1,

```
1 lc_101 $      ./unit_1/
2 unit_1 $
```

.. now refers to lc101.../.. here refers to launchcode_courses.

Like . (*Current Directory* (.)), .. isn't a command itself, but rather a notation. We're now ready to tackle our first command!

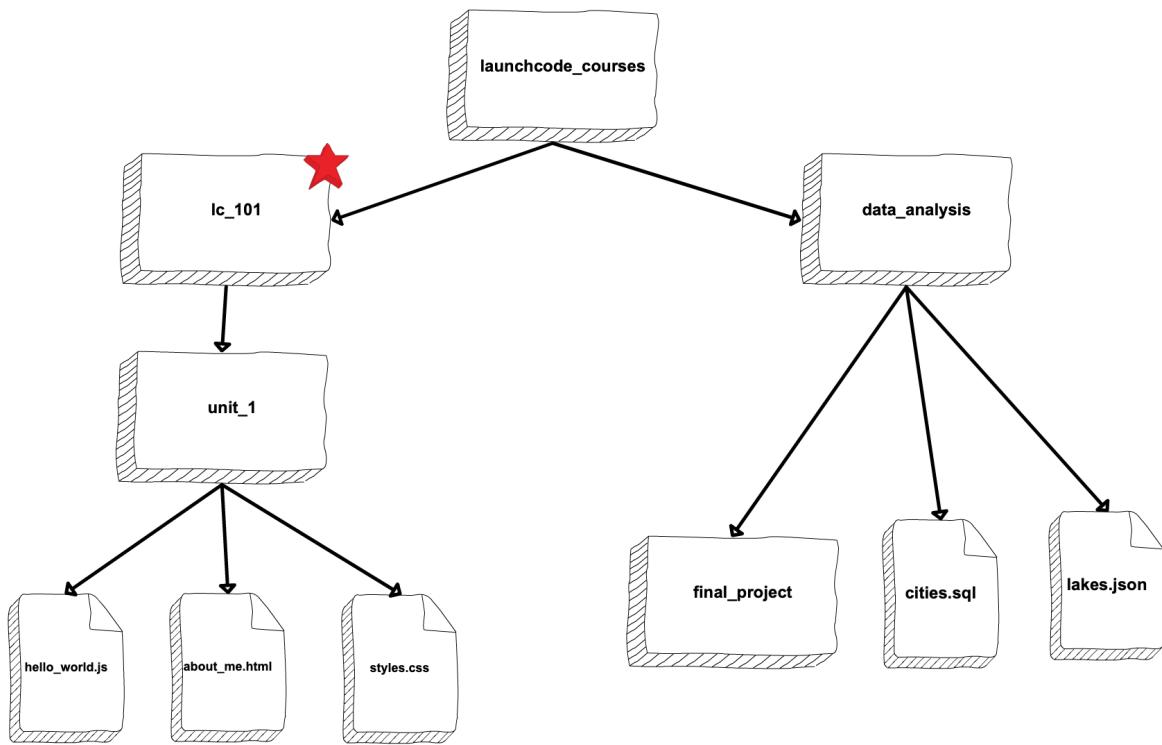


Fig. 4: We're now in lc_101

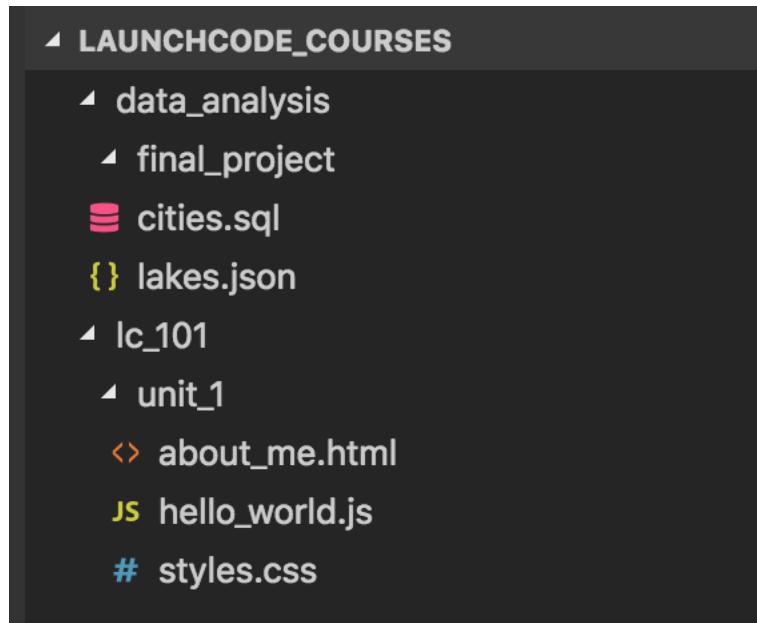


Fig. 5: launchcode_courses contains data_analysis and lc_101.

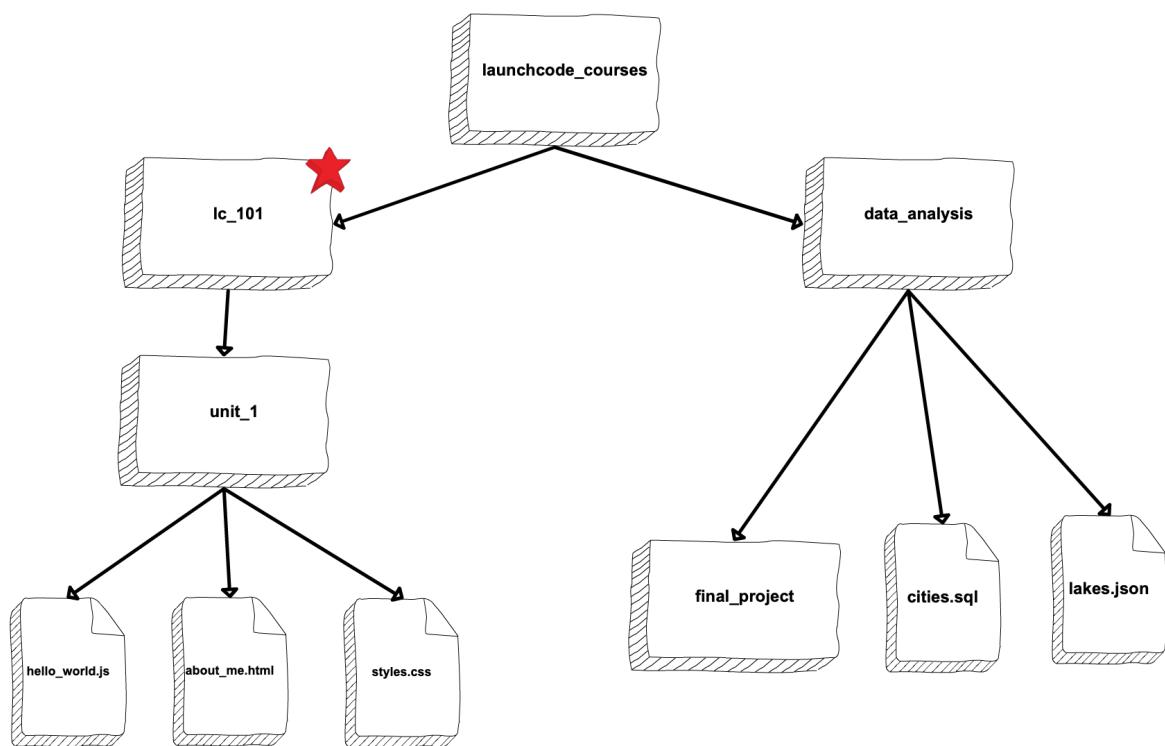


Fig. 6: We're still in `lc_101`.

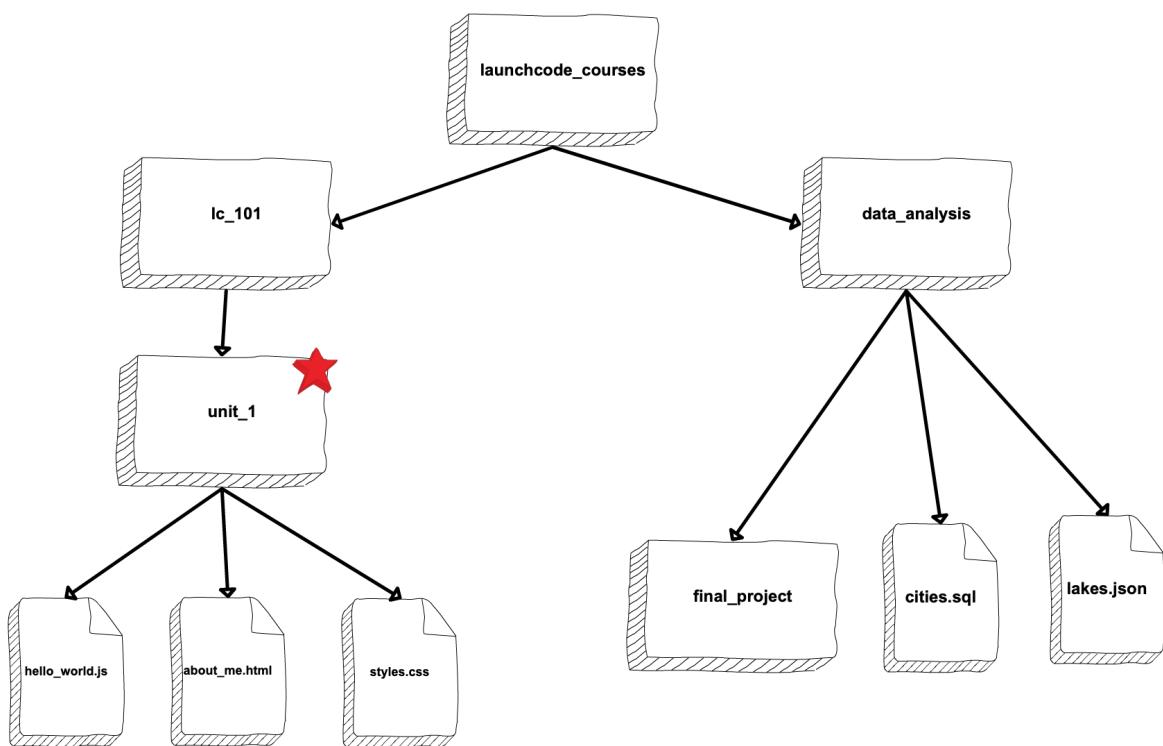


Fig. 7: We're now in unit_1.

pwd Command

Entering the `pwd` command in your terminal returns your current location, aka your **working directory**.

```
1 unit_1 $  
2 /launchcode_courses/lc_101/unit_1  
3 unit_1 $
```

The working directory is another term for the current directory. Think of this command as like the ‘You are here’ star on our file maps.

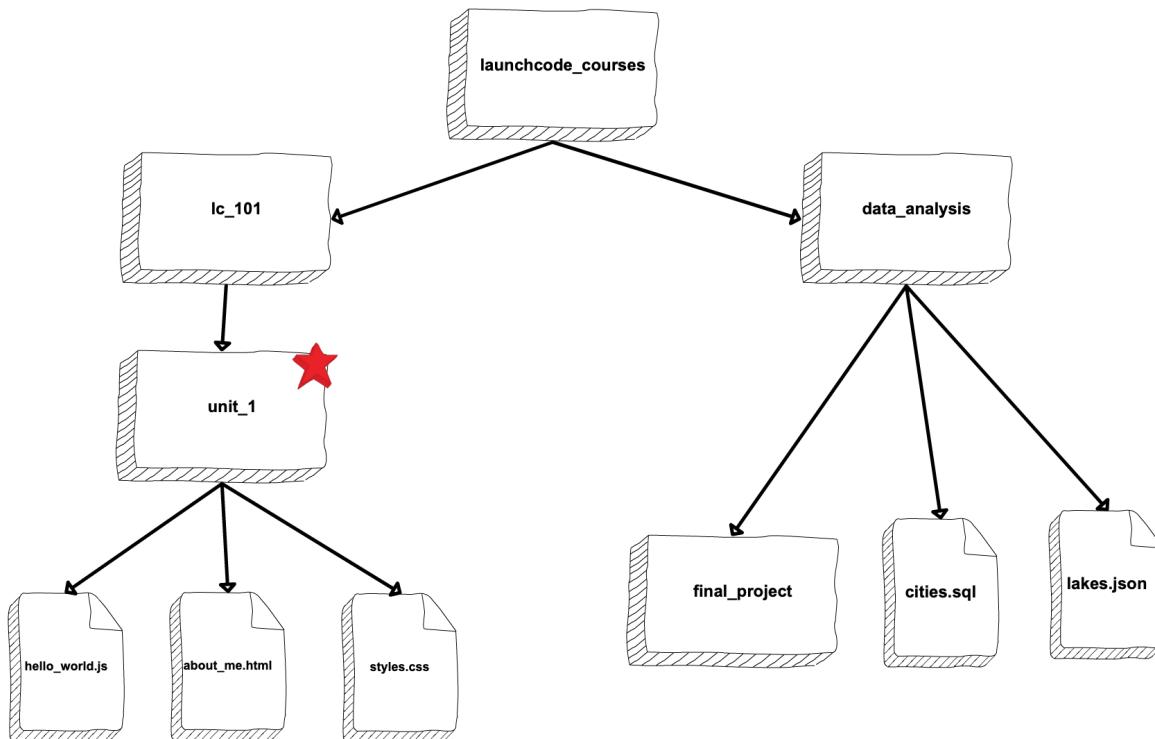


Fig. 8: We’re still in `unit_1`.

You’re basically just telling the computer to give you your current location. This may seem basic, but this one is essential. *You need to know your current location when working in the terminal.* A lot of beginner programmers simply enter commands into the terminal without mind to where they are. `pwd` is like a sanity check - a quick way to ensure that you know where you are and what you’re doing. It’s the file system counterpart to Git’s `git status`.

ls Command

Entering the `ls` command in your terminal returns the contents of your current directory. Recall, we’re in `unit_1`.

```
1 unit_1 $  
2 /launchcode_courses/lc_101/unit_1  
3 unit_1 $ ls  
4 about_me.html    hello_world.js    styles.css
```

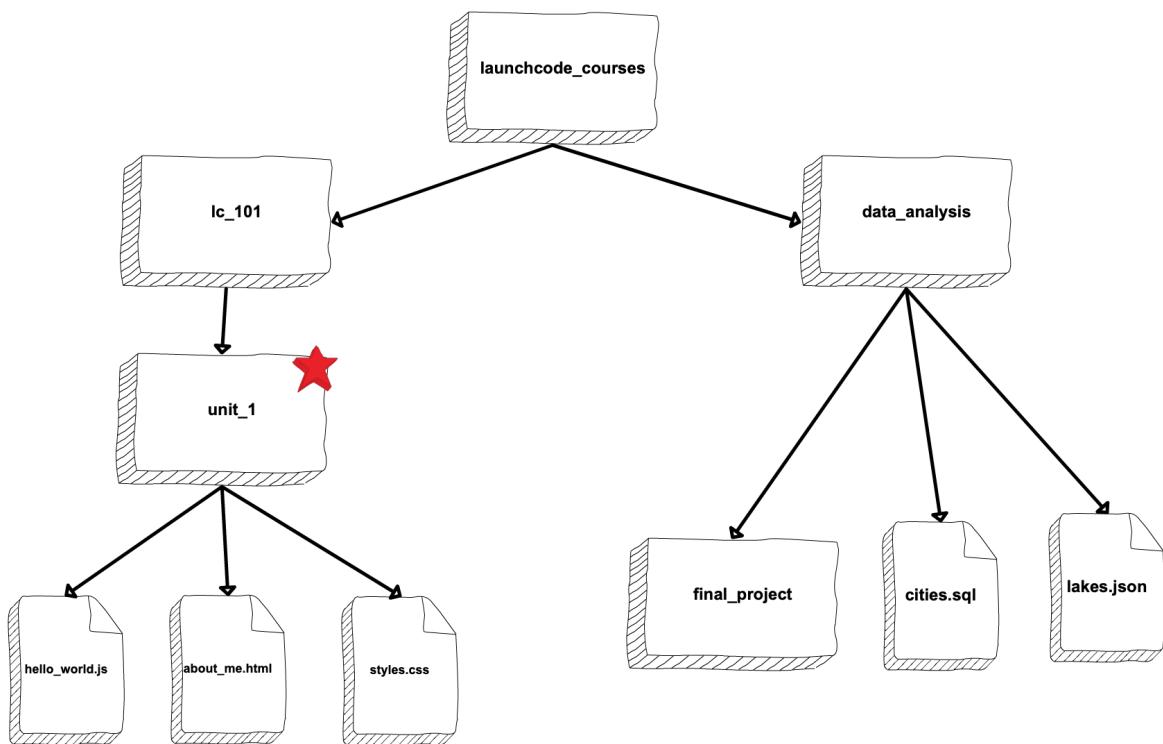


Fig. 9: We're still in `unit_1`.

All of that looks to be in order. Let's move back out into lc_101 and run ls from there.

```

1 unit_1 $
2 /launchcode_courses/lc_101/unit_1
3 unit_1 $ ls
4 about_me.html    hello_world.js  styles.css
5 unit_1 $ ..
6 lc_101 $
7 /launchcode_courses/lc_101
8 lc_101 $ ls
9 unit_1
10 lc_101 $
```

Notice that *pwd Command* after we moved. Also pay attention that *ls* only gives us a view one level deep. Now let's talk about how we move between directories.

cd Command

cd <path_name> relocates you to the provided path. We've seen it before, now let's explore this command some more.

Remember, we're inside lc_101,

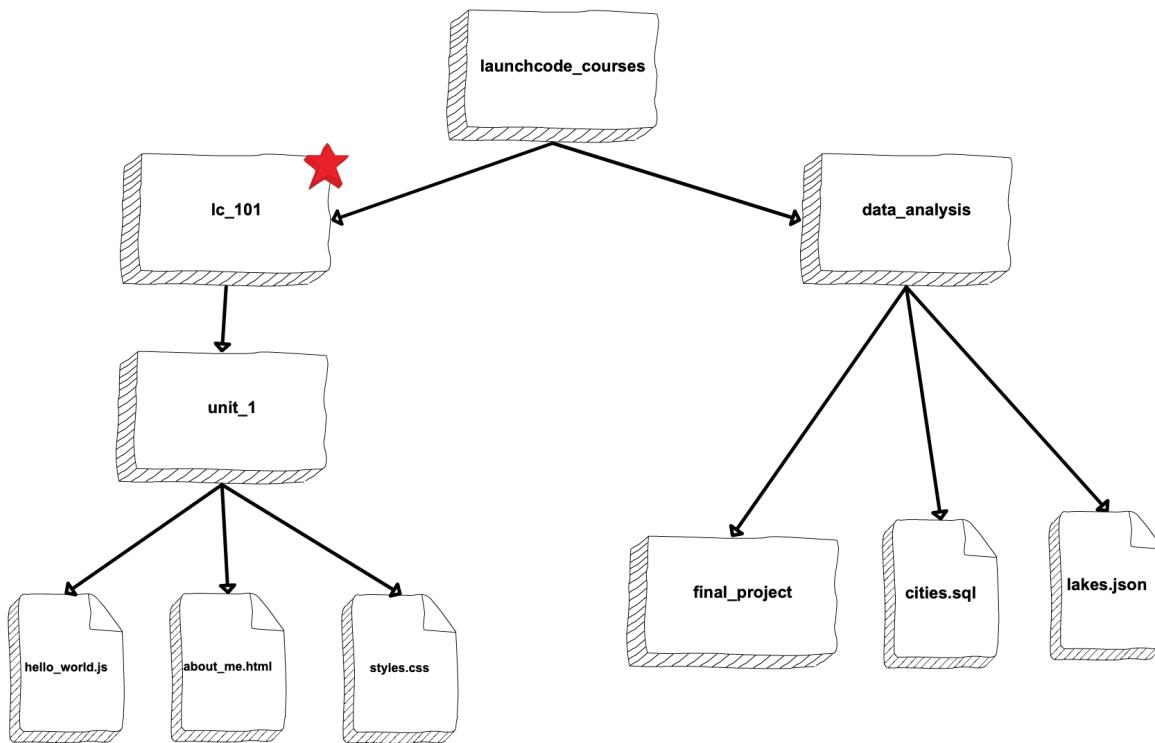


Fig. 10: We're in lc_101.

To change directories to our *Parent Directory* (..), we run the following:

```
1 lc_101 $  
2 /launchcode_courses/lc_101  
3 lc_101 $ ..  
4 launchcode_courses $  
5 /launchcode_courses  
6 launchcode_courses $
```

It's pretty self-explanatory, now we're back in `launchcode_courses`.

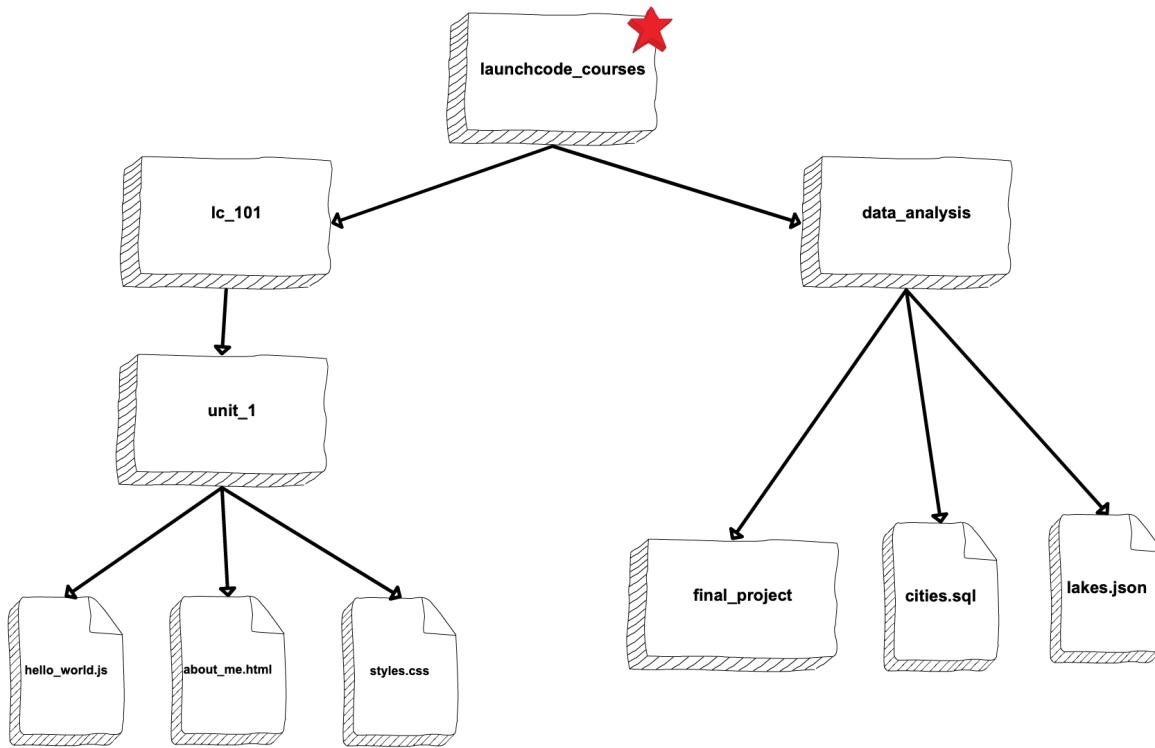


Fig. 11: We're back to `launchcode_courses`.

Not surprisingly, to go down into `data_analysis`, we run `cd ./data_analysis/`

```
1 launchcode_courses $  
2 /launchcode_courses  
3 launchcode_courses $ ./data_analysis/  
4 data_analysis $  
5 /launchcode_courses/data_analysis  
6 data_analysis $
```

Ok, so we know how to move one level above our current location (into our parent directory) and how to move one level below our working directory. But what if we wanted to get back to `lc_101` from where we are now, in `data_analysis`?

In order to move to a directory that is contained within the same parent as our working directory, we need to first go back up into the parent.

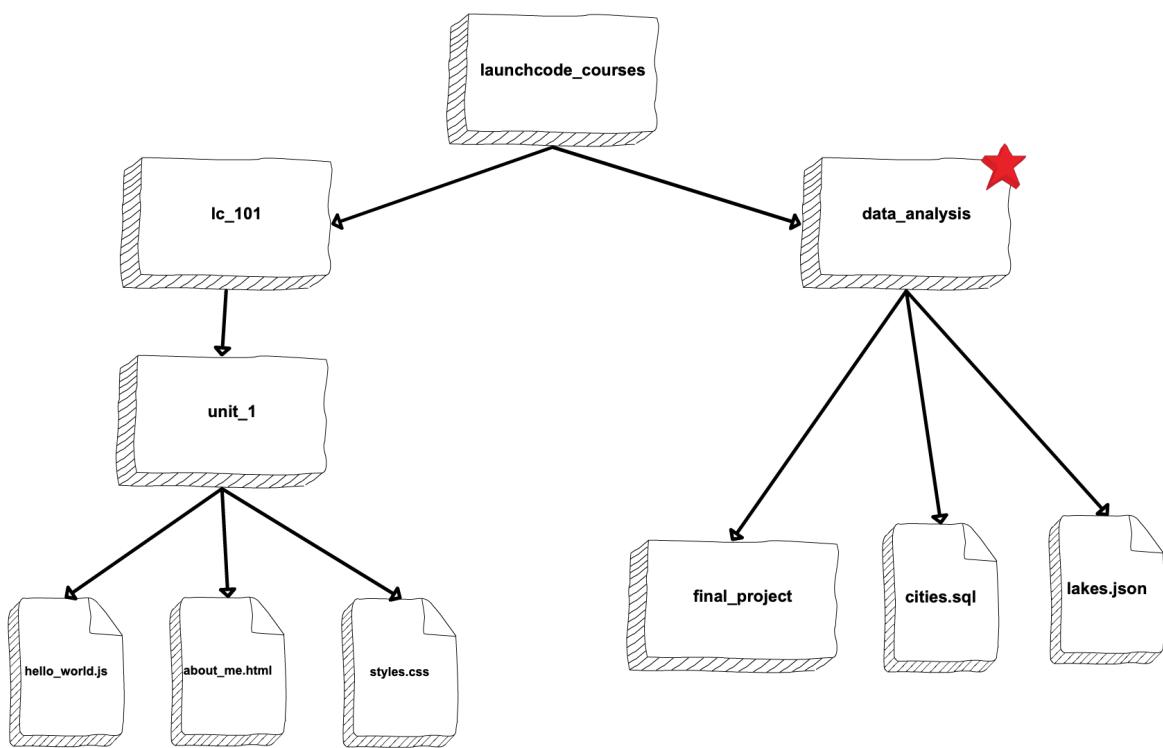


Fig. 12: We've made it to `data_analysis`.

```
1 data_analysis $  
2 /launchcode_courses/data_analysis  
3 data_analysis $ lc_101  
4 bash: cd: lc_101: No such file or directory  
5 data_analysis $  
6 /launchcode_courses/data_analysis  
7 data_analysis $ ..lc_101/  
8 lc_101 $  
9 /launchcode_courses/lc_101  
10 lc_101 $
```

Do you see the faulty command? We tried running `cd lc_101` from inside `data_analysis` but the terminal did not recognize that path name from inside the `data_analysis` directory.

We already know how to move to a parent directory, `cd ..`, above we see how we can move into a parent directory and down into one of its children all in one command, `cd ../lc_101/`.

Here's a visual of where we've just been

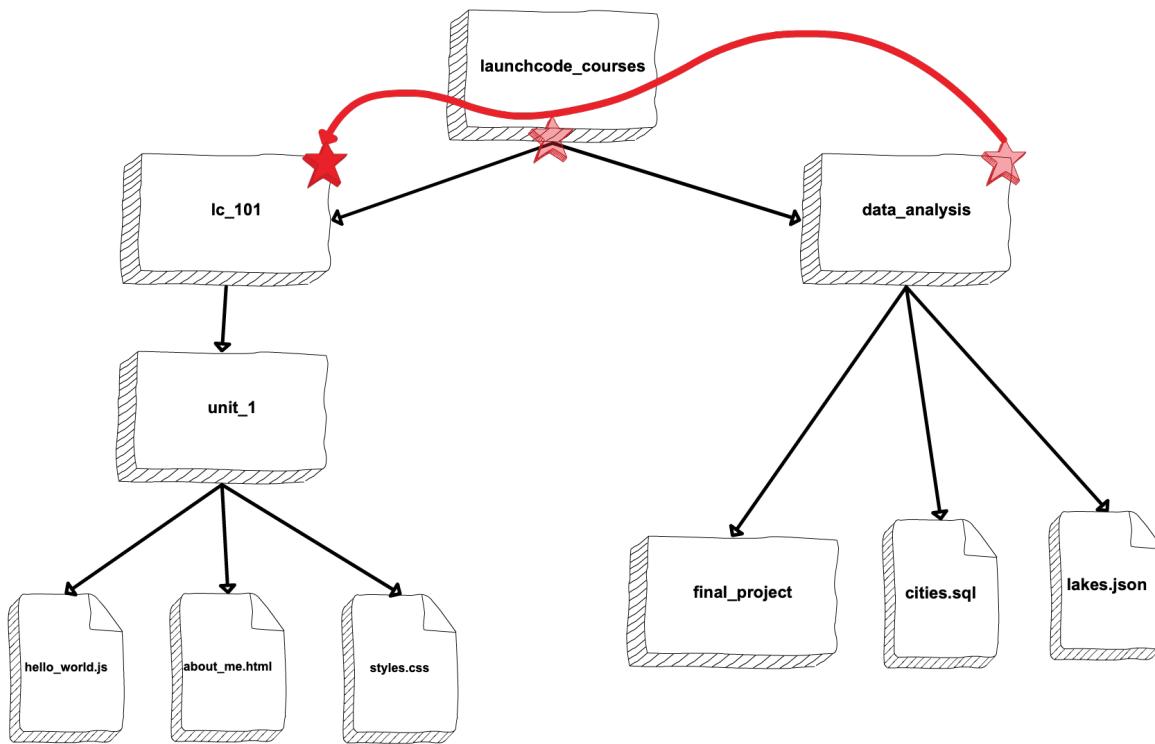


Fig. 13: Path to move to a peer directory.

For practice, let's go from our current spot in `lc_101`, down into `final_project`.

```
1 lc_101 $  
2 /launchcode_courses/lc_101  
3 lc_101 $ ..  
4 launchcode_courses $  
5 /launchcode_courses
```

(continues on next page)

(continued from previous page)

```
6 launchcode_courses $ ls
7 data_analysis lc_101
8 launchcode_courses $     data_analysis/
9 data_analysis $ ls
10 cities.sql final_project lakes.json
11 data_analysis $     final_project/
12 final_project $
13 launchcode_courses/data_analysis/final_project
14 final_project $
```

Above, we check our location as we navigate to make sure we know where we're going. If we're really confident though, we can accomplish moving from `lc_101` to `final_project` all in one go. Let's say we moved back to `lc_101` already.

```
1 lc_101 $
2 /launchcode_courses/lc_101
3 lc_101 $     ..../data_analysis/final_project/
4 final_project $
5 launchcode_courses/data_analysis/final_project
6 final_project $
```

Are you starting to see how terminal navigation can get you places swiftly?

Let's do one more quick move for fun. To go back to `lc_101`, all we need to do is `cd ../../lc_101/`.

```
1 final_project $
2 launchcode_courses/data_analysis/final_project
3 final_project $     ../../lc_101/
4 lc_101 $
5 launchcode_courses/lc_101
6 lc_101 $
```

Perhaps you noticed that the computer does not return anything to you after a successful `cd` command. In the navigation samples above, we frequently rely on the *pwd Command* and the *ls Command* to remind us where we are and what paths are available to us.

mkdir Command

`mkdir <new_directory_name>` creates a new directory *inside* your current location.

We're in the `lc_101` directory.

Here, let's create a directory for Unit 2 materials.

```
1 lc_101 $
2 launchcode_courses/lc_101
3 lc_101 $ ls
4 unit_1
5 lc_101 $ mkdir unit_2
6 lc_101 $ ls
7 unit_1   unit_2
8 lc_101 $
```

Again, the computer does not return anything to you after this command and simply responds ready to accept another prompt. But we can see from our helpful *ls Command* that a new directory has been created.

And we can visualize our changes like this:

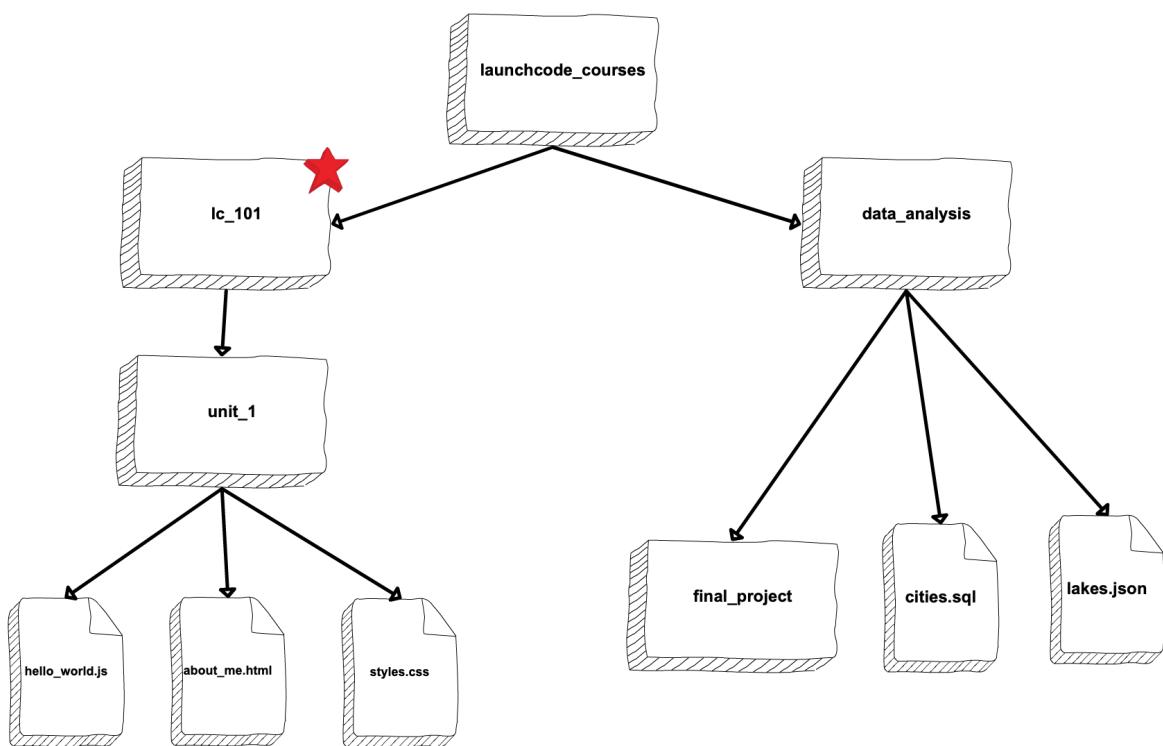


Fig. 14: We're back in lc_101.

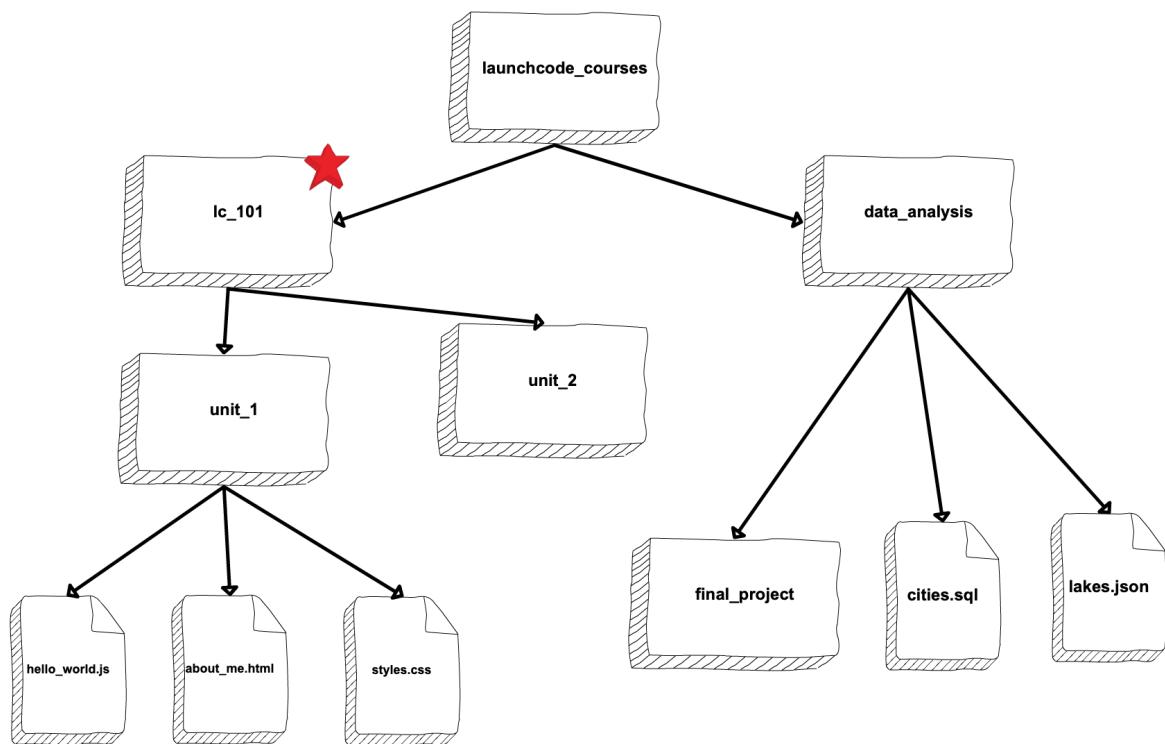


Fig. 15: mkdir creates a new directory

Note

While `mkdir` creates a new directory, it does not place us into that directory. Additionally, we don't need to be in the parent of the newly created directory. We can run `mkdir` from anywhere within the file system, as long as we use the appropriate file path.

`rm` Command

`rm <item_to_remove>` removes a given item from the file tree.

Let's say we decide we no longer need our `cities.sql` data. We can remove it!

For fun - and practice! - let's remove it while we're still located in the `lc_101` directory.

```
1 lc_101 $  
2 launchcode_courses/lc_101  
3 lc_101 $ rm ../data_analysis/cities.sql  
4 lc_101 $  
5 launchcode_courses/lc_101  
6 lc_101 $ ls ../data_analysis/  
7 final_project    lakes.json  
8 lc_101 $
```

See what we did there? Instead of moving into the parent directory of `cities.sql`, we just used the longer file path relative to our location in `lc_101`. And to check that our `rm` command did what we expected? Well we also checked that right from our spot in `lc_101` with `ls` and a longer path.

Here's the map of what we've done:

To remove a directory entry, rather than simply a file, requires an **option** on the command. An option is an additional character, or set of characters, added on the end of a text command to give the computer more instructions related to your command. Options are usually indicated with a `-`. We'll talk more about the presence of options in *man Command*.

A common method to remove a directory is to use the `-r` option, although there are other choices.

Let's say we no longer want our `unit_2` directory. We're still in `lc_101`.

```
1 lc_101 $ ls  
2 unit_1  unit_2  
3 lc_101 $ rm unit_2  
4 rm: unit_2: is a directory  
5 lc_101 $ ls  
6 unit_1  unit_2  
7 lc_101 $ rm -r unit_2  
8 lc_101 $ ls  
9 unit_1  
10 lc_101 $
```

Notice, we try using simply `rm` but we get a response returned that the item we've asked to remove is a directory. But alas, with `rm -r`, we are able to successfully remove the `unit_2` directory.

Back in our map:

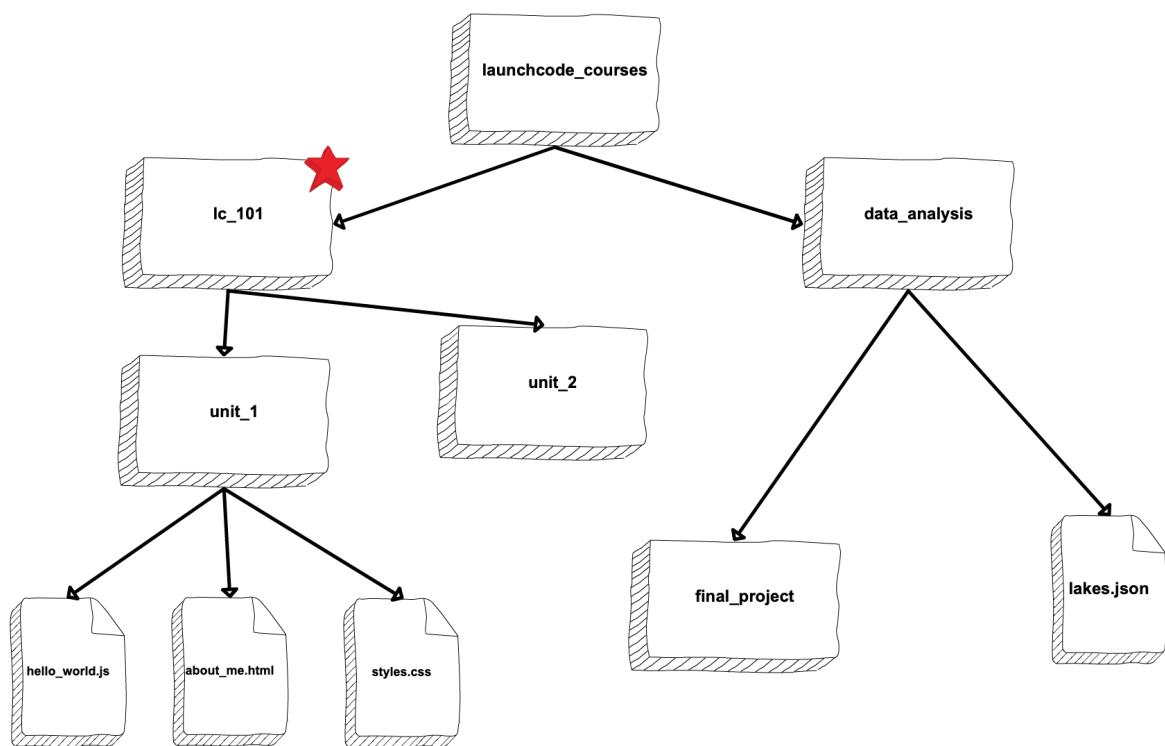


Fig. 16: `cities.sql` is gone!

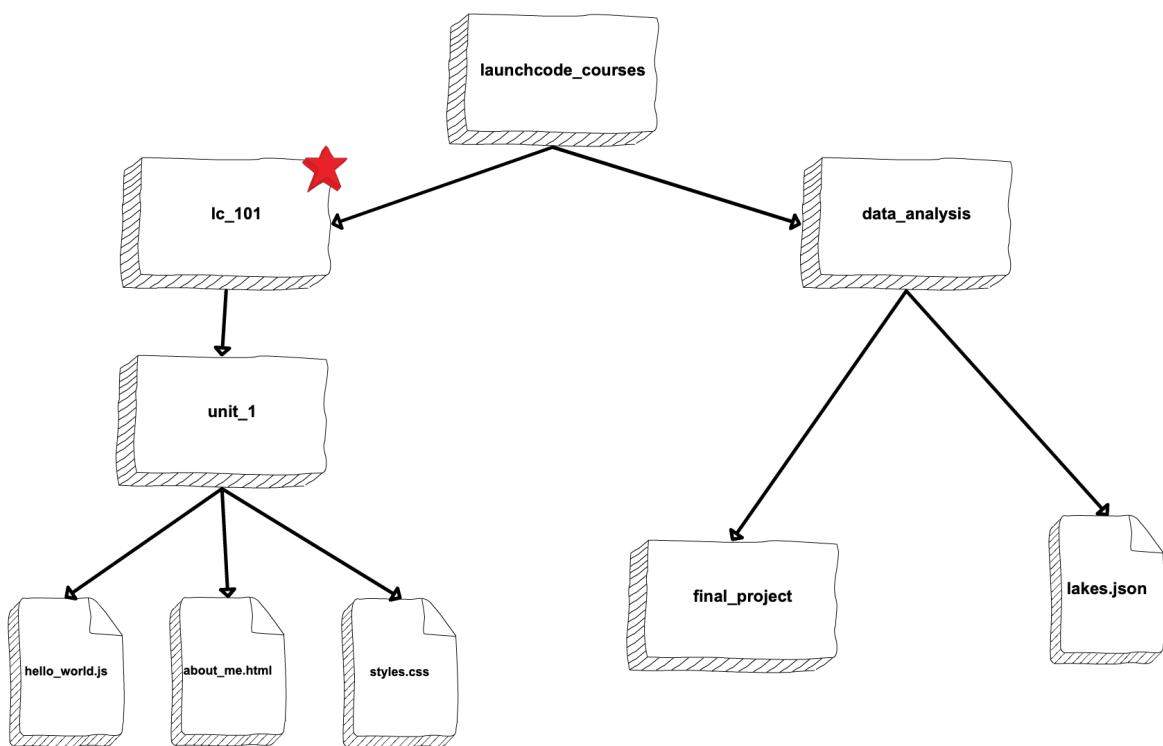


Fig. 17: unit_2 is gone without a trace

cp Command

`cp <source_path> <target_path>` copies the item at the source and puts it in the target path. The item can be a file or whole directory and is named within its own source path.

Take our sample file tree above. We're still in `lc_101` and say we want to copy our `lakes.json` file and place that copy inside the `final_project` directory.

```
1 lc_101 $  
2 launchcode_courses/lc_101  
3 lc_101 $     ./data_analysis/  
4 data_analysis $  
5 launchcode_courses/data_analysis  
6 data_analysis $ ls  
7 final_project    lakes.json  
8 data_analysis $ cp ./lakes.json ./final_project/  
9 data_analysis $ ls  
10 final_project   lakes.json  
11 data_analysis $ ls ./final_project/  
12 lakes.json  
13 data_analysis $
```

We didn't need to cd into `data_analysis` but since we are dealing with a file contained within it, it made sense to do so. Once we ran our `cp` command, we checked the contents of both `data_analysis` and `data_analysis/final_project` to verify the copy was made.

And of course, now there are two lakes.json.

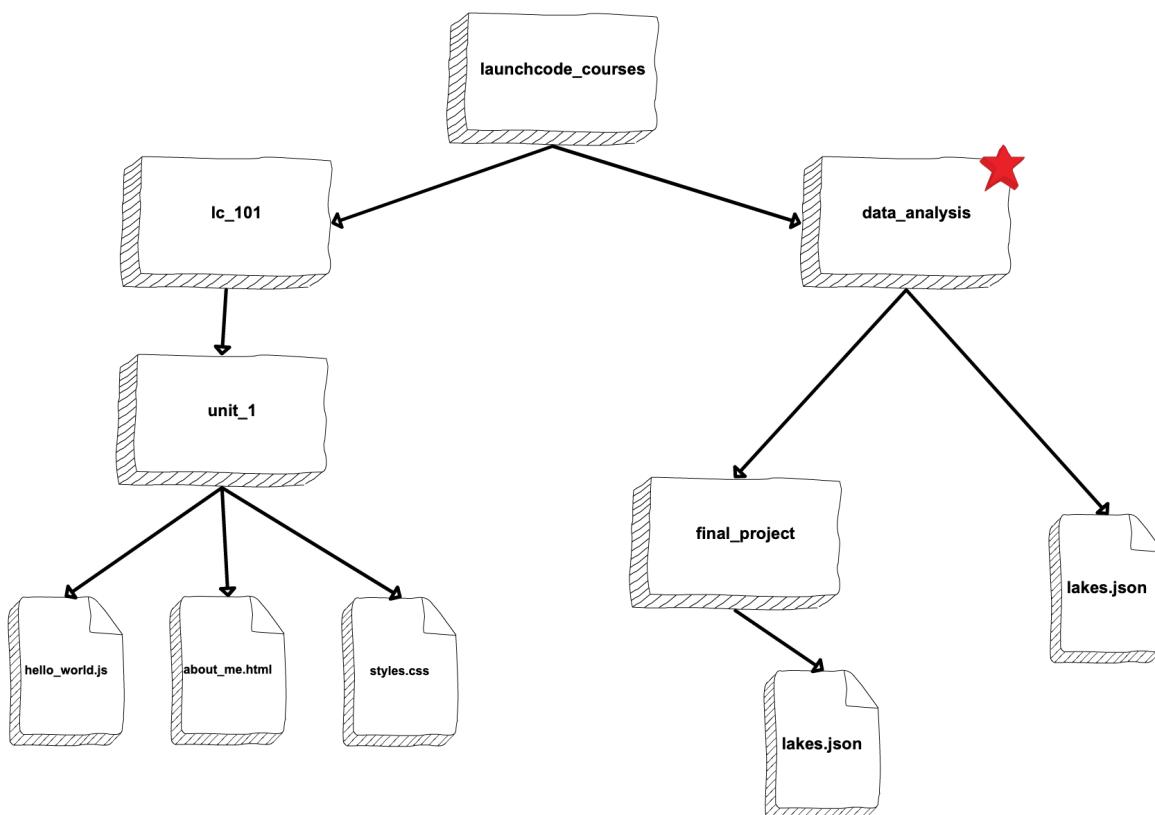


Fig. 18: lakes.json double take

We can think of `cp` as basically copy *and* paste, since the target path is included in the command.

`mv` Command

`mv <item_to_move> <target_path>` moves an item to the provided target path. The item being moved can be a single file or a whole directory. When referring to the item being moved, its source path is required, just like the `cp` Command.

Still in `data_analysis`, lets move `data_analysis/lakes.json` into `lc_101`.

```
1 data_analysis $ mv ./lakes.json ../../lc_101/
2 data_analysis $
3 launchcode_courses/data_analysis
4 data_analysis $ ls
5 final_project
6 data_analysis $ ls ../../lc_101/
7 lakes.json    unit_1
8 data_analysis $
```

As usual, we use `ls` to verify our results. Now our map looks like the following:

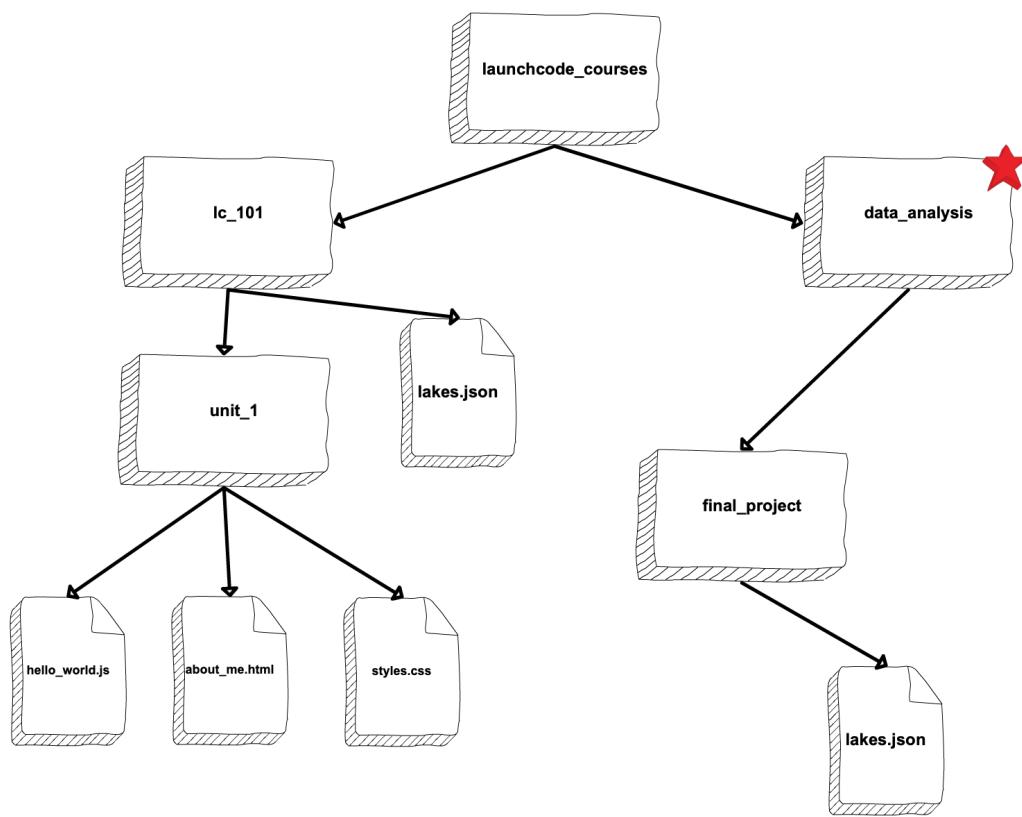


Fig. 19: `mv` moves one of the `lakes.json`.

`touch` Command

`touch <new_file_name>` creates a new file.

Back in `data_analysis`, lets add a new `cafes.sql` file to our directory.

```

1 data_analysis $
2 launchcode_courses/data_analysis
3 data_analysis $ ls
4 final_project
5 data_analysis $ touch cafes.sql
6 data_analysis $ ls
7 cafes.sql    final_project
8 data_analysis $
```

Here's what that gives us:

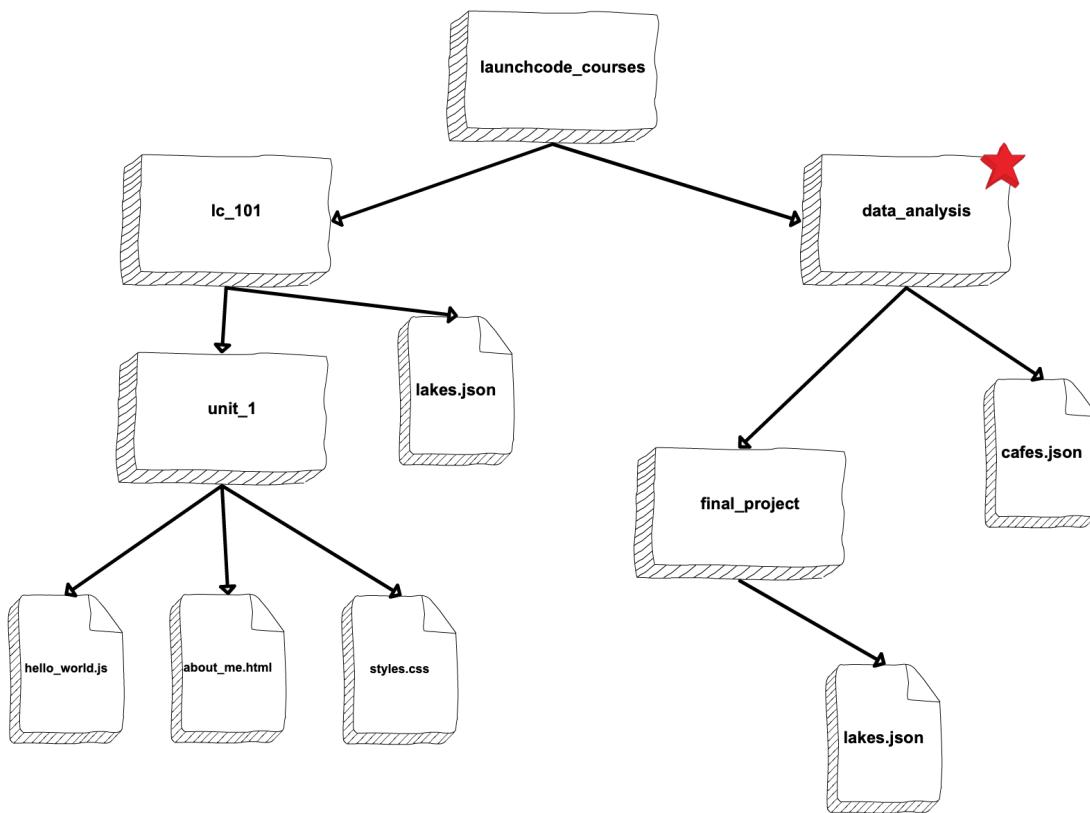


Fig. 20: `touch` adds a file

clear Command

`clear` wipes your terminal window of any previously run commands and outputs in case you need a clean screen to think straight.

You probably won't encounter a scenario where you *need* to clear your terminal, but it can be a nice command to know if you're a minimalist.

There's no change to our file map to show when this command is run. And in the terminal window, as soon as enter is hit, the command results in what looks like a new window.

```

1 data_analysis $
```

man Command

man is your best friend. Running `man <command>` gives you a manual entry of what that command does, what options it takes, and more documentation than you could ever need. It's so thorough, it makes this guide blush. Any command you think you may need, but you're not sure how to use it, or maybe you want to do something specific and are wondering if there's a specialized option for it, use `man` to get more info!

Practice looking up some of the commands you know; maybe you'll learn a new option or two!

Some other terminal stuff you should know when using the manual:

- **Scrolling** Some entries are very long! They will probably need to be scrolled through. You'll know there's more to read if you see `:` at the bottom of your terminal window. You can use your keyboard's arrow keys to navigate the entry. If you reach the bottom of the entry, you'll see a line that reads `END`.
- **Exiting** Once you're finished reading, you'll need to exit the manual page using the `q` command.

Exiting Programs

ctrl + c Details

`ctrl + c` can be used to exit a running program.

Some programs take different commands to exit. `ctrl + c` is sometimes the command to quit a running program and other times used to prompt the running program for a different exit command.

`q`

`q` is another command for exiting a running program. Notably, it is needed to exit the *man Command* pages.

35.10 Setting up Software for the Class

35.10.1 Setting Up Your Terminal

The command line is a vital tool to learn for any programmer. Over the course of this program and your career, you will find yourself navigating to it frequently.

Note

Once you have your terminal setup on your machine, make sure to pin it to the taskbar or add it to your dock!

Mac Users

Good news! The Terminal application comes with every Mac.

You can access it one of two ways:

Through the Finder

1. Open a new Finder window and navigate to the Applications folder.
2. Inside the Applications folder, you will find inside a Utilities folder.
3. Open the Utilities folder and inside is the Terminal application!

Through LaunchPad

1. If you are a fan of the LaunchPad features on Apple computers, hit the F4 key.
2. Inside the Other or Utilities folder, you can find the Terminal.

If you are still struggling to find the Terminal application, you can do a simple search in the Finder for it!

Windows Users

1. In order to get your CLI up and running, you have to first install [Git Bash](#).

Note

When you are doing your Git Bash setup, you only need to leave the default selected.

2. Once Git Bash is installed on your machine, you can find the folder for it through the Home screen.
3. Inside the folder, simply select Git Bash to open the appropriate CLI.

35.11 TODOs

Todo: add exercises for fetch

(The original entry is located in /Users/chris/dev/launchcode/lc101/intro-to-professional-web-dev/src/chapters/fetch-json/exercises.rst, line 88.)

35.12 Chapters

35.12.1 Studios

35.12.2 Assignments

35.12.3 Appendices

HTML Me Someting, Part 1

In Part 1, you will get comfortable with writing markup, and with separating *content* from *design and layout*.

Getting Started

Stub out `index.html` with these basic elements:

```
1 <!DOCTYPE html>
2
3   html
4     head
5       title  title
6     head
7
8   body   body
9   html
```

Fill each element with some appropriate content. You can start by removing the snippet `<p>YOUR NAME</p>` that you used during setup.

Getting to Work

Your mission is to build a page that:

1. Tells a story. This can be personal or impersonal, funny, serious or neither. You can do whatever you like, but it should be something in the range of 3-10 paragraphs or sections. [Here is an example](#), and here are some other ideas:
 - a. Create a résumé page that tells the story of your professional journey to-date, and where you want to go as a coder.
 - b. Describe a trip you took.
 - c. Talk about one of your hobbies or passions.
2. Does each of the following:
 - a. Uses each of the following structural HTML5 tags: `<p>`, `<header>`, `<footer>`, `<main>`, `<article>`. If you need to review any of these tags, check out the [HTML tag reference](#) at w3schools.
 - b. Uses at least one `` tag (hopefully more). When placing images in your page, put them in a new subfolder called `images` within your `html-me-something` directory.
 - c. Uses at least one [HTML entity](#). Hint: putting a copyright notice in your footer will afford you the opportunity to use `©`, but you should also try to get creative here.
 - d. Demonstrates creativity. Don't stop with these items or tags. Have some ideas for your page, and make it great. And dig into the [w3schools HTML reference](#) to learn more about other tags, their usage and attributes!

Notes and Tips

1. Use your browser developer tools to look at [the example page](#), or to troubleshoot things that don't look right. You can mimic the document structure of this example, but do NOT just copy/paste!
2. Use the example to learn how your HTML elements might be structured, and build your page to fit your own content.
3. Don't add any CSS yet. Really, we mean it! If you think your page looks boring now, that's okay. We'll get there soon enough.

4. As you make changes, you will obviously want to see the results. To do so, re-save the file in your text editor, then click Refresh in the browser window (or use cmd+R on a Mac, ctrl+R on Windows).
5. Rely on the reference sites linked on this page, or find others online. We haven't taught you every detail about every tag that you may want or need.
6. You're free to use tags that haven't been explicitly introduced in class. We've given you enough background to get started, and are intentionally leaving some of the learning up to you.

Add and Commit Your Changes on Git

Once you finish your page, use Git to add and commit your `index.html` changes.

Why commit again?

The reason is that you added a bunch of new HTML code to your `index.html` file. It is now in a very different state compared to the first time you committed it. See this for yourself by checking the status:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Git tells us:

I see that you have modified `index.html`. Use `git add` if you want these changes to be included in your next commit.

The Git workflow more or less comes down to this:

1. Create some initial stuff
2. init
3. add and commit
4. make some changes
5. add and commit
6. make some more changes
7. add and commit
8. etc...

The general rule is that *whenever you make any changes, you must add and commit those changes to Git*.

So let's do that. From within your `html-me-something/` directory:

```
$ git add index.html
$ git commit -m "Finished work on HTML page"
```

You might be wondering: *How do I know when it's time to pause working and do another commit?*

This is somewhat subjective, and is ultimately up to you. The good habit is to pause and commit any time you reach a natural stopping point or complete a coherent chunk of work.

Done!

Well done! Time to [dive](#) into some CSS.

HTML Me Something, Part 2

In Part 2, you'll get comfortable with using CSS selectors and rules to dictate display, while keeping your styles separate from your content.

Getting Started

1. Create a file named `styles.css` in your `html-me-something/` directory.
2. *Optional:* Add a normalization stylesheet (follow any of the links in the *Normalization* section below). You can either put these normalization rules at the top of your `styles.css`, or you can add another file in the same directory and link to it in your HTML doc. This will reset some of your browser's built-in styles so that you start with a cleaner slate when you add your own.

Getting to Work

Go ahead and start adding styles in your `styles.css` file!

Requirements

Your CSS must:

1. Use `margin` and `padding` to space your elements in a visually pleasing way.
2. Use at least one of each of the following types of selectors:
 - a. `element`,
 - b. `class`,
 - c. `id`.
3. Follow these rules:
 - a. Avoid adding HTML elements in order to achieve a specific visual effect.
 - b. Use document-level and inline styles sparingly, and only when absolutely necessary.
 - c. Be creative! Make your page look great, and don't just settle for checking off the items above. Have a look at [CSS Zen Garden](#) for inspiration (use your browser's `developer tools` to see how those pages' styles are built).

Notes

1. In order to see any visible change, make sure to link the stylesheet to your main document.
2. Feel free to check out our [styled example](#) to see how we did things. Use "View Source" (by right-clicking anywhere on the page and selecting *View Source*).
3. Another thing you might find useful is your browser's `developer tools`.
4. And be sure to use the *Resources* section below as you go.

Resources

General CSS:

1. [w3schools CSS Reference](#)
2. [CSS Zen Garden](#)
3. [\(Advanced\) Specifics on CSS Specificity](#)
4. [\(Advanced\) Specificity \(MDN\)](#)

CSS Normalization:

1. [Eric Meyer's reset.css](#)
2. [normalize.css](#)

Add and Commit Your Changes on Git

When you finish making your page look spiffy, take a moment to commit your changes on Git:

```
$ git add styles.css  
$ git commit -m "Added some killer CSS styling"
```

If you also made tweaks to your `index.html` file, then you need to commit those changes as well (along with a descriptive commit message):

```
$ git add index.html  
$ git commit -m "Changed title from 'My Favorite Puppies' to 'The Objectively Best_Puppies'"
```

Incidentally, this is a good opportunity to address the question: *Why does Git have two separate commands for “add” and “commit” if I always do them together anyway?*

The answer comes back to the notion of collecting your changes in a “coherent chunk of work” for each commit. The `add` command gives you the opportunity to specify exactly *which* file(s) should be included in the upcoming commit. In the example above, this allowed us to perform two separate commits—each with its own message describing its own chunk of work.

Note that you certainly can `add` multiple files to the same commit. For example, suppose you made changes to both `index.html` and `styles.css`, but those changes are all part of the same unit of work. In that case you would add them both before committing:

```
$ git add index.html  
$ git add styles.css  
$ git commit -m "Added puppy image with thick yellow border"
```

There is a convenient shortcut, `git add .`, for those (frequent) occasions when you want to include *every* changed file without having to type each one individually. The following example is equivalent to the previous one (assuming you only changed `index.html` and `styles.css`):

```
$ git add .  
$ git commit -m "Added puppy image with thick yellow border"
```

It is usually a good idea to check first (using `git status`) before running `git add .`, so that you don't mistakenly include unwanted changes.

Done!

You are ready to submit! Go back to the *Assignment Page* and follow the submission instructions there.