LaunchCode logo

Search

Introduction to Professional Web Development in JavaScript

- Summary
- Expanded

Chapters¶

Studios¶

Assignments¶

Appendices¶

Back to top

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. Introduction
- 1.2. Why Learn JavaScript? →

Back to top

LaunchCode logo

Search

1. Contents
2. 1. Introduction
3. 1.2. Why Learn JavaScript?

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.

- ← 1.1. Why Learn To Code?
- 1.3. About LaunchCode Programs →

Back to top

LaunchCode logo

Search

1. Contents
2. 1. Introduction
3. 1.3. About LaunchCode Programs

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. Course Activities¶

We have created the course activities to make the most of your time. It is important to actively engage with each activity to maximize your learning potential. Skipping the textbook reading or falling behind on assignments can quickly lead to struggling to complete the course.

1.3.2.1. Textbook Reading¶

Think of this textbook as your first destination in your learning journey. In addition to reading, this text includes small questions that can help you reinforce your understanding of the new material. Reviewing your notes from a given chapter before moving on to the next is another great way to make the most of your learning potential.

1.3.2.2. Exercises¶

At the end of most of the textbook chapters, you will find a a page of *Exercises*. These 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.

1.3.2.3. Graded Assignments¶

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

- ← 1.2. Why Learn JavaScript?
- 1.4. Blended Learning →

Back to top

LaunchCode logo

Search

1. Contents
2. 1. Introduction
3. 1.4. Blended Learning

1.4. Blended Learning¶

This page covers what to expect during in-class time for students enrolled in a LaunchCode blended learning course. Students taking an independent learning course may skip this page.

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 course. A blended learning model incorporates in-class learning with online materials like this textbook.

1.4.1. In-Class Time¶

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

In-class time is run by an instructor and teaching assistants. This dedicated staff facilitates the activities and provides support to the students.

1.4.1.1. 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 reading.

1.4.1.2. Small Group Time¶

After the large group time, we break up into small groups, each led by a teaching assistant. During small group time, we do in-class coding activities called studios. This is a time to ask for individual support if you need it. It is meant to be a place where you can feel comfortable talking openly about concepts you are struggling with.

- ← 1.3. About LaunchCode Programs
- 1.5. Class Platforms →

Back to top

LaunchCode logo

Search

1. Contents
2. 1. Introduction
3. 1.5. Class Platforms

1.5. Class Platforms¶

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

1.5.1. 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.

1.5.1.1. Login to Canvas¶

Access Canvas and the course materials at https://launchcode.instructure.com/. To login, use your launchcode.org username and password, which are the same ones you used to apply for this class.

1.5.1.2. Canvas Dashboard¶

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

Canvas sign-in and dashboard views

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

Canvas class menu options

1.5.1.3. Syllabus Page¶

The syllabus page provides general information, such as a description of the class, the timeline for the course, a calendar, and a todo 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 list

1.5.1.4. Assignments Page¶

This page sorts required tasks by date or type. We regularly add new tasks to this list so check back here often. Old content remains active, allowing you to use the links for reference and review.

Course assignment view

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

Sample task instructions

Even though much of the course content can be accessed without logging in, the best choice is to begin your course work 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.

1.5.2. 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 provides opportunities to respond to prompts, questions, "Try It" exercises, and studios embedded within the reading. These tasks are neither tracked nor scored.

1.5.2.1. Repl.it Account Creation¶

Creating a Repl.it account is covered in Chapter 2. You'll need to create an account to use Repl.it.

1.5.2.2. Online Code Editor¶

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

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

Tip

You never have to click save when using Repl.it. Repl.it automatically saves your code on their servers.

1.5.3. GitHub Classroom¶

GitHub Classroom provides online code storage. For this class, it also allows for graded assignment submission and evaluation.

GitHub is a web application that uses *version control*. We'll learn more about GitHub and what version control is in a future lesson.

Note

Results from work submitted in GitHub 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.

- ← 1.4. Blended Learning
- 1.6. Using This Book →

Back to top

1.6. 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.

1.6.1. Concept Checks¶

Many pages end with a "Check Your Understanding" header. 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.

1.6.2. 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!

1.6.2.1. Repl.it¶

As we mention on the previous page, we expect you to use your **Repl.it** account to practice your programming skills.

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.

1.6.3. Supplemental Content¶

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

1.6.4. 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.

- ← 1.5. Class Platforms
- 2. How Programs Work →

Back to top

LaunchCode logo

Search

1. Contents
2. 2. How Programs Work

2. How Programs Work¶

- ← 1.6. Using This Book
- 2.1. Introduction →

Back to top

LaunchCode logo

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:

1. Answering a math problem.
2. Sorting numbers in decreasing order.
3. Making a peanut butter and jelly sandwich.
4. Assigning guests to tables at a wedding reception.
5. Creating a grocery list.
6. Suggesting new music for a playlist.
7. Making cars self-driving.

- ← 2. How Programs Work
- 2.2. Programming Languages →

Back to top

LaunchCode logo

Search

1. Contents
2. 2. How Programs Work
3. 2.2. Programming Languages

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.

2.2.1.1. 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.

Visual description for how a compiler works.

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.

- ← 2.1. Introduction
- 2.3. The JavaScript Language →

Back to top

LaunchCode logo

Search

1. Contents
2. 2. How Programs Work
3. 2.3. The JavaScript Language

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¶

Difference between front and back end changes.

- ← 2.2. Programming Languages
- 2.4. Your First Program →

Back to top

LaunchCode logo

Search

1. Contents
2. 2. How Programs Work
3. 2.4. Your First Program

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!

Note

Below is a guide to walk you through setting up a Replit account, creating a new repl, and working with repls made by other people. The images are there to help guide you, but Replit.com may update or change how it looks. Don't hesitate to explore the site on your own.

2.4.1. Create a Replit 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 Replit.com. The site provides a free space to practice coding.

After you have created your account, 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.

Note

If your dashboard is completely empty, click on the "Hamburger" button in the top left corner. The three stacked horizontal lines. This will reveal a drop-down menu that allows you to create new repls, access your saved repls, etc.

2.4.1.1. Creating a New Repl¶

From your dashboard, click on + *Create* or any of the + buttons to open the "Create a repl" window.

Replit new repl

Steps to create a new repl:

1. You need to **select the coding language template** you want to work with. Select "Node.js" to code in JavaScript.
2. **Name your project** to help you remember what the repl contains.
3. **Click** "Create Repl".

These steps will create a blank repl workspace where you can start programming in JavaScript.

2.4.1.2. The Replit Workspace¶

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

Replit code editor layout

Features to note:

1. **File panel and menus**: This 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. **Run button**: Executes any code written in the `index` file.

Note

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

2.4.2. Begin Your Coding Journey¶

2.4.2.1. Working with a Prepared Repl¶

We have prepared a repl for you. You will not be allowed to code into it since you did not create it. To add your own code, you will need to **fork** it. Forking creates a copy of the original codebase into your repl account. You can code in the forked project without risk of changing the original codebase.

Replit fork option

Follow this Hello World link to open a prepared workspace for your first program.

Your forked copy will open in your workspace. On line 2 of the editor panel, type:

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

When you finish typing, click the green "Run" button and observe the output in the console panel.

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.

2.4.2.2. 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.

1. `console.log(Coding Rocks)`
2. `console.log(Coding Rocks);`
3. `console.log('Coding Rocks')`
4. `console.log("Coding Rocks');`
5. `console.log("Coding Rocks");`

- ← 2.3. The JavaScript Language
- 3. How To Write Code →

Back to top

LaunchCode logo

Search

1. Contents

2. 3. How To Write Code

3. How To Write Code¶

- ← 2.4. Your First Program
- 3.1. What is Code? →

Back to top

LaunchCode logo

Search

1. Contents
2. 3. How To Write Code
3. 3.1. What is 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. How To Write Code
- 3.2. Syntax Rules →

Back to top

LaunchCode logo

Search

1. Contents
2. 3. How To Write Code
3. 3.2. Syntax 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.

- ← 3.1. What is Code?
- 3.3. Comments →

Back to top

LaunchCode logo

Search

1. Contents
2. 3. How To Write Code
3. 3.3. Comments

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 or interpreter. 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 `/*` and `*/`.

Try It

Experiment by adding and removing comments to the code.

```
// This demo shows off comments!

// console.log("This does not print.");

console.log("Hello, World!"); // Comments do not have to start at the b

/* Here is how
to have
multi-line
comments. */

console.log("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!

- ← 3.2. Syntax Rules
- 3.4. Output With `console.log` →

Back to top

LaunchCode logo

Search

1. Contents
2. 3. How To Write Code
3. 3.4. Output With `console.log`

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 console.log('Hello, JavaScript.');
2 console.log(2001);
3 console.log("What","do","commas","do?");
4 console.log("Does", "adding",      "space", "matter?");
5 console.log('Launch' + 'Code');
6 console.log("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 four words, separated by commas, all within the same set of parentheses `()`. When these four words print, they show up on the same line but separated by spaces.
4. The code in line 4 puts extra 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* is 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 console.log("Some Programming Languages:");
2
3 console.log("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`.

- ← 3.3. Comments
- 3.5. Fixing Errors in Your Code →

Back to top

LaunchCode logo

Search

1. Contents
2. 3. How To Write Code
3. 3.5. Fixing Errors in Your Code

3.5. Fixing Errors in Your Code¶

As you progress through this class, you will find yourself having to fix problems in your code. Whether the code simply won't run or is giving you the wrong answer, fixing problems with code is a vital part of being a programmer. The process of working through issues in code is called **debugging**. We will cover debugging in depth in a later chapter, but here is what we need to know now.

1. Errors are common and even the most experienced programmers are regularly debugging their code.
2. Asking for help from peers and mentors can be a key part of the process.
3. Google is very much an acceptable resource when it comes to debugging.

3.5.1. Asking for Help¶

Before asking for help, make sure that you have the answers to some of the questions that may be asked of you and that your questions to the person who is helping you are detailed. "My code doesn't work" or "Why doesn't my code work?" are difficult questions for someone to answer. "I tried using `console.log()` like the chapter outlined, but the screen says 'Invalid Input'. What does that mean and how might I go about addressing it?" is a good question to ask.

Here are some tips on what information you might need when asking for help:

1. Ask Google for help first. If you see an error message as opposed to "Hello World!", copy it into Google and search. You may find the answer right away! If you don't find the answer or are not sure you understand it, make note of that.
2. Make sure to document everything. The steps you took before you encountered the problem, screenshots of what the error message is, what your application should be able to do. This is all helpful information for the person assisting you.
3. When asking your question, make sure that the person knows what documentation you have. If you took a screenshot or saved Google results related to the error message, inform the person who is helping you before you all start working together on the issue.

When trying to fix an error in code, many people start by following the exact same steps to see if they get the same results. The more information you can give the person who is helping you, the easier it will be for them to help you.

Note

These tips also apply to whenever you encounter a technical issue. Having screenshots and a recorded set of steps that you took before encountering

the issue can help tech support figure out how to correct the problem and assist you!

- ← 3.4. Output With `console.log`
- 3.6. Welcome, Novice Coder →

Back to top

LaunchCode logo

Search

1. Contents
2. 3. How To Write Code
3. 3.6. Welcome, Novice Coder

3.6. 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.

- ← 3.5. Fixing Errors in Your Code
- 4. Data and Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables

4. Data and Variables¶

Back to top

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 console.log(typeof "Hello, World!");
2 console.log(typeof 17);
3 console.log(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 console.log(typeof "17");
2 console.log(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 console.log(typeof 'This is a string');
2 console.log(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 console.log(42000);
2 console.log(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 console.log(42, 17, 56, 34, 11, 4.35, 32);
2 console.log(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 languages 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 number 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. Data and Variables
- 4.2. Type Conversion →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.2. Type Conversion

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 console.log(Number("2345"));
2 console.log(typeof Number("2345"))
3 console.log(Number(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

```
console.log(Number("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 console.log(String(17));
2 console.log(String(123.45));
3 console.log(typeof String(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'`

- ← 4.1. Values and Data Types
- 4.3. Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.3. Variables

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.

A label, programmingLanguages, pointing to a the string value "JavaScript"

A variable can be visualized as a label pointing to a specific 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`:

```
let programmingLanguage;
```

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 programmingLanguage;
2 programmingLanguage = "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.

The name "programmingLanguage" with no arrow connecting it to data.

The result of `let programmingLanguage;`¶

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

A label, programmingLanguages, pointing to a the string value "JavaScript"

The result of `programmingLanguage = "JavaScript";`¶

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.

```
let programmingLanguage = "JavaScript";
```

Warning

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

```
var programmingLanguage = "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" = programmingLanguage;
```

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?

```
programmingLanguage = "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 output.

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

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

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

```
let message = "What's up, Doc?";
let n = 17;
let pi = 3.14159;

console.log(message);
console.log(n);
console.log(pi);
```

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

```
let message = "What's up, Doc?";
let n = 17;
let pi = 3.14159;

console.log(typeof message);
console.log(typeof n);
console.log(typeof pi);
```

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.

```
let day = "Thursday";
console.log(day);

day = "Friday";
console.log(day);

day = 21;
console.log(day);
```

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 day = "Thursday";
2 day = 32.5;
3 day = 19;
4 console.log(day);
```

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 a;
2 a = 42;
3 a = a + 3;
```


- ← 4.2. Type Conversion
- 4.4. More On Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.4. More On Variables

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 programmingLanguage = "JavaScript";
2 programmingLanguage = "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.

A to-do list web application with application name in the title bar and main header.

An example to-do list web application¶

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 appName = "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 appName = "Get It Done!";
```

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 appName = "Get It Done";
2 appName = "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¶

4.4.2.1. 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 application name;
```

Console Output

```
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.

4.4.2.2. 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 x = 5;
2 const y = 3.14;
3 let z = y * x ** 2;
4 console.log(z);
```

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 of this program.

```
1 let radiusOfCircle = 5;
2 const pi = 3.14;
3 let areaOfCircle = pi * radiusOfCircle ** 2;
4 console.log(areaOfCircle);
```

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.

4.4.2.3. 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 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)

- ← 4.3. Variables
- 4.5. Expressions and Evaluation →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.5. Expressions and Evaluation

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

```
console.log(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 sum = 1 + 2;
2 console.log(sum);
```

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`.

- ← 4.4. More On Variables
- 4.6. Operations →

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 console.log(2 + 3);
2 console.log(2 - 3);
3 console.log(2 * 3);
4 console.log(2 ** 3);
5 console.log(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 minutes = 645;
2 let hours = minutes / 60;
3 console.log(hours);
```

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.

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 (`**`)		`3 ** 2` returns `9`

Operator	Description	Example
	Calculates the base (first operand) to the exponent (second operand) power, that is, base^{exponent}	`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 `(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 console.log(2 ** 3 ** 2)
3
4 // use parentheses to force the order you want
5 console.log((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. 13.666666666666666

Question

What is the output of the code below?

```
console.log(1 + 5 % 3);
```

Question

What is the value of the following expression?

```
2 ** 2 ** 3 * 3
```

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

- ← 4.5. Expressions and Evaluation
- 4.7. Other Operators →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.7. Other Operators

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

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

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

Example

```
1 console.log(1 + 1);
2 console.log("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 x = 1;
2 x = x + 1;
3
4 console.log(x);
```

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 x = 1;
2 x += 1;
3
4 console.log(x);
```

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**.

Compound Assignment Operators¶

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

- ← 4.6. Operations
- 4.8. Input with `readline-sync` →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.8. Input with `readline-sync`

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 input = require('readline-sync');
2
3 let info = input.question("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.

4.8.2.1. 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.

4.8.2.2. 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 input = require('readline-sync');
2
3 let name = input.question("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 input = require('readline-sync');
2
3 let num1 = input.question("Enter a number: ");
4 let num2 = input.question("Enter another number: ");
5
6 console.log(num1 + num2);
```

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 input = require('readline-sync');
2
3 let info = input.question("Please enter your age: ");
4 //The user enters 25.
5
6 console.log(typeof info);
```

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

- ← 4.7. Other Operators
- 4.9. Exercises: Data and Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.9. Exercises: Data and Variables

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.

We've included our solutions to some (but not all) of the exercises so that you can check your own work after you practice. As with any learning opportunity, take it seriously and give every exercise a fair shot before peeking at the solution. Our solution may not look exactly like yours, but that's ok. There's often more than one way to solve a problem with code.

You don't need to show proof of completing the exercises to pass, so do them for your own benefit. And if they're hard or you get some answers wrong on the first try, keep going! Your future self will thank you.

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.

Use this Repl.it link to get started solving.

4.9.1. The Data¶

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

4.9.2. The Exercises¶

1. **Declare and assign variables**

   Declare and assign a variable for each item in the list above.

   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!

   Check your solution.

2. **Print out the type of each variable**

   For each variable you declared in part A, use the `typeof` operator to print its type to the console, one item per line.

   Click "Run" (in repl.it) and verify that your code works before moving to part C.

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.

   Check your solution

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 miles to Mars by the shuttle's speed.

   Check your solution

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.

   Check your solution

4. **Print out the results of your calculations**

   Using variables from 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.

   Click "Run" (in repl.it) and verify that your code works before moving on.

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."`.

   Check your solution

- ← 4.8. Input with `readline-sync`
- 4.10. Studio: Data and Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 4. Data and Variables
3. 4.10. Studio: Data and Variables

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.

4.10.1. Before You Start¶

Use this starter Repl.it file to complete this studio.

4.10.2. Declare and Initialize Variables¶

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

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

4.10.3. 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 *).

4.10.3.1. Example Output¶

Note that your output should 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%

-------------------------------------
> MASS 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
```

4.10.4. 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.

4.10.5. 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).

- ← 4.9. Exercises: Data and Variables
- 5. Making Decisions With Conditionals →

Back to top

LaunchCode logo

Search

1. Contents
2. 5. Making Decisions With Conditionals

5. Making Decisions With Conditionals¶

- ← 4.10. Studio: Data and Variables

- 5.1. Booleans →

Back to top

LaunchCode logo

Search

1. Contents
2. 5. Making Decisions With Conditionals
3. 5.1. Booleans

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 console.log(true);
2 console.log(typeof true);
3 console.log(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 console.log(typeof true);
2 console.log(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 console.log(Boolean("true"));
2 console.log(Boolean("TRUE"));
3 console.log(Boolean(0));
4 console.log(Boolean(1));
5 console.log(Boolean(''));
6 console.log(Boolean('LaunchCode'));
```

repl.it

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 console.log(5 == 5);
2 console.log(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

```
console.log(true == "true");
```

Console Output

```
false
```

5.1.3.1. Comparison Operators¶

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

Comparison Operators¶

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

Operator	Description	Examples Returning `true`	Examples Returning `false`
		`7 >= 7`	
	hand operand, and `false` otherwise.	`'b' >= 'a'`	`'a' >= 'b'`
		`'b' >= 'b'`	

Less than or equal (<=)	Returns `true` if the left-hand operand is less than or equal to the right-hand operand, and `false` otherwise.	`5 <= 7`
		`5 <= 5`
		`'a' <= 'b'`
		`'a' <= 'a'`
			`7 <= 5`
			`'b' <= 'a'`

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

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`.

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. Making Decisions With Conditionals
- 5.2. Equality →

Back to top

LaunchCode logo

Search

1. Contents
2. 5. Making Decisions With Conditionals
3. 5.2. Equality

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 console.log(7 == "7");
2 console.log(0 == false);
3 console.log(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 console.log('0' == 0);
2 console.log(0 == '');
3 console.log('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 console.log(7 === "7");
2 console.log(0 === false);
3 console.log(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.

- ← 5.1. Booleans
- 5.3. Logical Operators →

Back to top

LaunchCode logo

Search

1. Contents
2. 5. Making Decisions With Conditionals
3. 5.3. Logical Operators

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.

5.3.1.1. 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 console.log(7 > 5 && 5 > 3);
2 console.log(7 > 5 && 2 > 3);
3 console.log(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.

5.3.1.2. 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 console.log(7 > 5 || 5 > 3);
2 console.log(7 > 5 || 2 > 3);
3 console.log(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 symbols 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 ||.

5.3.1.3. Logical NOT¶

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

Example

```
1 console.log(! true);
2 console.log(! false);
```

Console Output

```
false
true
```

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

Example

```
1 console.log( !(5 > 7) );
2 console.log( !('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:

```
((x * 5) >= 10) && ((y - 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.

Operator Precedence¶

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

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

Truth Table for &&¶

A	B	A && B
`true`	`true`	`true`

A	B	A && B
`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.

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"`

- ← 5.2. Equality
- 5.4. Conditionals →

Back to top

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:

The structure of a conditional with an if statement

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

```
let billHasBeenPaid = false;

if (!billHasBeenPaid) {
   console.log("Your bill is due soon!");
}
```

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

```
if (num % 2 === 0 && num > 3) {
   console.log(num, "is even");
   console.log(num, "is greater than 3");
}
```

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.

```
if (num % 2 === 0 && num > 3)
{ console.log(num, "is even");
 console.log(num, "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 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.

```
if (!billHasBeenPaid)
   console.log("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.

A conditional with an else clause

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

Example

```
let billHasBeenPaid = true;

if (!billHasBeenPaid) {
   console.log("Your bill is due soon!");
} else {
   console.log("Your payments are up to date.");
}
```

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`.

A diagram showing how the flow of a program branches based on the value of the condition in an if-else statement. If the condition is true, one code block executes. If the condition is false, a different code block executes.

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 x = 10;
 2 let y = 20;
 3
 4 if (x > y) {
 5    console.log("x is greater than y");
 6 } else if (x < y) {
 7    console.log("x is less than y");
 8 } else {
 9    console.log("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 x = 10;
2 let y = 10;
3
4 if (x > y) {
5    console.log("x is greater than y");
6 } else if (x < y) {
7    console.log("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

```
let x = 10;
let y = 20;

if (x > y) {
   console.log("x is greater than y");
} else if (x < y) {
   console.log("x is less than y");
} else if (x % 5 === 0) {
   console.log("x is divisible by 5");
} else if (x % 2 === 0) {
   console.log("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?

```
let a = 7;
if (a % 2 === 1) {
   console.log("Launch");
} else if (a > 5) {
   console.log("Code");
} else {
   console.log("LaunchCode");
}
```

1. `"Launch"`
2. `"Code"`

"Launch"
3. "Code"
4. "LaunchCode"

- ← 5.3. Logical Operators
- 5.5. Nested Conditionals →

Back to top

LaunchCode logo

Search

1. Contents
2. 5. Making Decisions With Conditionals
3. 5.5. Nested Conditionals

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:

```
let num = 7;

if (num % 2 === 0) {
   console.log("EVEN");
}

if (num > 0) {
   console.log("POSITIVE");
}
```

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.

```
let num = 7;

if (num % 2 === 0) {
    console.log("EVEN");

    if (num > 0) {
        console.log("POSITIVE");
    }
}
```

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?

```
let num = 7;

if (num % 2 === 0) {
    if (num % 2 === 1) {
        console.log("odd");
    }
}
```

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?

```
if (num % 2 === 0) {
    if (num % 2 === 1) {
        console.log("odd");
    }
}
```

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.

- ← 5.4. Conditionals
- 5.6. Exercises: Booleans and Conditionals →

Back to top

LaunchCode logo

Search

1. Contents
2. 5. Making Decisions With Conditionals
3. 5.6. Exercises: Booleans and Conditionals

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.

Take note of any problem here or topic from this chapter that you don't understand. Take a break and return to the problem. Do you see it in a new way or have a better understanding? If not, try spending five minutes researching the topic. Start with this book and if you still have questions, ask one on the internet or Slack/Discourse. You're not the first person to learn to code and you're definitely not the first person to ask a question or get stuck!

Code exercises A & B here.

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

Variable Name	Value
engineIndicatorLight	red blinking
spaceSuitsOn	true

Variable Name	Value
shuttleCabinReady	true
crewStatus	spaceSuitsOn && shuttleCabinReady
computerStatusCode	200
shuttleSpeed	15000

Check your solution.

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

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

```
if (engineIndicatorLight === "green") {
   console.log("engines have started");
} else if (engineIndicatorLight === "green blinking") {
   console.log("engines are preparing to start");
} else {
   console.log("engines are off");
}
```

Code exercises C & D here.

1. **Write conditional expressions to satisfy the safety rules.**

   Use the variables defined from the table above to satisfy the rules listed below.

   1. `crewStatus`

      - If the value is `true`, print `"Crew Ready"`
      - Else print `"Crew Not Ready"`

      Check your solution

   1. `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!"`

      Check your solution

   2. `shuttleSpeed`

      - If the value is `> 17500`, print `"ALERT: Escape velocity reached!"`
      - Else if the value is `< 8000`, print `"ALERT: Cannot maintain orbit!"`
      - Else print `"Stable speed"`

Check your solution

2. **PREDICT**

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

```
1 if (crewStatus && computerStatusCode === 200 && spaceSuitsOn) {
2   console.log("all systems go");
3 } else {
4   console.log("WARNING. Not ready");
5 }
```

```
1 if (!crewStatus || computerStatusCode !== 200 || !spaceSuitsOn) {
2   console.log("WARNING. Not ready");
3 } else {
4   console.log("all systems go");
5 }
```

Code exercises E & F here.

1. **Monitor the shuttle's fuel status.**

   Implement the checks below using `if` / `else if` / `else` statements. Order is important when working with conditionals, and the checks below are NOT written in the correct sequence. Please read ALL of the checks before coding and then decide on the best order for the conditionals.

   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.

fuelLevel	engineTemperature	engineIndicatorLight	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.`

Check your solution

1. **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!"`

- ← 5.5. Nested Conditionals
- 5.7. Studio: Goal Setting and Getting into the Right Mindset →

Back to top

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

- ← 5.6. Exercises: Booleans and Conditionals
- 6. Errors and Debugging →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging

6. 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 1947, 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. Errors and Debugging
- 6.2. Categories of Errors →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging
3. 6.2. Categories of Errors

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.

6.2.1.1. 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.

6.2.1.2. 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 day = Wednesday;
2 console.log(day;
```

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 firstName = "Jack";
2 console.log(firstname);
```

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/fs/streams.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

```
let weeklyPay = 600;

let dailyEarnings = weeklyPay / 7;
console.log(dailyEarnings);
```

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.

- ← 6.1. What is Debugging?
- 6.3. Diagnosing Error Messages →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging
3. 6.3. Diagnosing Error Messages

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

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.

A screenshot with two lines of code. Syntax errors on each line cause highlighting to differ from what is expected. On line 1, the string "Julie" is green instead of red, because it is missing quotes. On line 2, the symbols); are red instead of black, because the preceding string "Hello, World" doesn't have a closing double-quote.

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.

- ← 6.2. Categories of Errors
- 6.4. Error Types →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging
3. 6.4. Error Types

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.

JavaScript Error Types¶

Error Type	Description	Example of code triggering the error	Example description
`SyntaxError`	Occurs when trying to parse syntactically invalid code.	`console.log("hello";`	The call to `console.log` doe not have a required close parenthesis.
`ReferenceError`	Occurs when a non-existent variable is used/ referenced.	`1 let firstName = "Jack";` `2 console.log(firstname);`	The variable `firstname` does not exist; it is a misspelling of `firstName`.
`TypeError`	Occurs when trying to use a value in an invalid way.	`1();`	The numeric valu 1 is not a functio so trying to use i as one results in `TypeError: 1 i not a function`
`RangeError`	Occurs when passing an invalid value to a function.	`let nums = Array(-1);`	The constructor function `Array(n` creates an empty array of length `n` It is not possible to create an arra with negative length, so the code results in `RangeError: Invalid array length`.
`URIError`	Occurs when improperly using a global URI-handling	`decodeURI('%');`	The `%` character used to encode characters not otherwise allowe in URIs, such as

Error Type	Description	Example of code triggering the error	Example description
	function. ('URI' = Uniform Resource Identifier)		spaces (`%20`). If a invalid character encoding is given a `URIError` results.
`Error`	The type from which all other errors are built. It can be used to generate programmer-triggered and programmer-defined errors.	`throw Error("Something bad happened!");`	Manually trigger 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!**

- ← 6.3. Diagnosing Error Messages
- 6.5. Debugging Logic Errors →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging
3. 6.5. Debugging Logic Errors

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 input = require('readline-sync');
2
3 let degreesC = input.question('Temp in degrees C:');
4 let degreesK = degreesC + 273.15;
5
6 console.log('Degrees K:', degreesK);
```

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 input = require('readline-sync');
2
3 let degreesC = input.question('Temp in degrees C: ');
4 console.log(degreesC);
5 let degreesK = degreesC + 273.15;
6
7 console.log('Degrees K:', degreesK);
```

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 input = require('readline-sync');
2
3 let degreesC = input.question('Temp in degrees C: ');
4 console.log(typeof degreesC);
5 let degreesK = degreesC + 273.15;
6
7 console.log('Degrees K:', degreesK);
```

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 input = require('readline-sync');
2
3 let degreesC = input.question('Temp in degrees C: ');
4 degreesC = Number(degreesC);
5 let degreesK = degreesC + 273.15;
6
7 console.log('Degrees K:', degreesK);
```

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.

- ← 6.4. Error Types
- 6.6. How to Avoid Debugging →

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.

- ← 6.5. Debugging Logic Errors
- 6.7. Asking Good Questions →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging
3. 6.7. Asking Good Questions

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 I 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."

- ← 6.6. How to Avoid Debugging
- 6.8. Exercises: Debugging →

Back to top

LaunchCode logo

Search

1. Contents
2. 6. Errors and Debugging
3. 6.8. Exercises: Debugging

6.8. Exercises: Debugging¶

Imagine we are running a space station. Your job is to evaluate the station's code and fix any errors. The lives of the crew rest squarely upon your shoulders.

Your directions from superiors:

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.
3. If all checks are successful, print a countdown to launch in the console.

6.8.1. Debugging Practice¶

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

```
1 let launchReady = false;
2 let fuelLevel = 17000;
3
4 if (fuelLevel >= 20000 {
5    console.log('Fuel level cleared.');
6    launchReady = true;
```

```
} else {
   console.log('WARNING: Insufficient fuel!');
   launchReady = false;
}
```

Fix it at repl.it

Check your solution.

2. The next block of code hides two syntax errors. Run the code as-is to find the mistakes.

```
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.");
}
```

Tip

Only one error will be flagged at a time. Fix that ONE problem, and then re-run the code to check your work. Avoid trying to fix multiple issues at once.

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 you locate and repair the mistake in the code.

```
let launchReady = false;
let fuelLevel = 17000;

```

```
if (fuellevel >= 20000) {
   console.log('Fuel level cleared.');
   launchReady = true;
} else {
   console.log('WARNING: Insufficient fuel!');
   launchReady = false;
}
```

Fix it at repl.it

Check your solution.

4. *Arrr!* Did we mention your crew are space pirates? Now find and fix the runtime error in a longer code sample.

```
let launchReady = false;
let fuelLevel = 27000;

if (fuelLevel >= 20000) {
   console.log('Fuel level cleared.');
   launchReady = true;
} else {
   console.log('WARNING: Insufficient fuel!');
   launchReady = false;
}

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

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.

```
let launchReady = false;
let fuelLevel = 17000;
let crewStatus = true;
```

```
let computerStatus = 'green';

if (fuelLevel >= 20000) {
   console.log('Fuel level cleared.');
   launchReady = true;
} else {
   console.log('WARNING: Insufficient fuel!');
   launchReady = false;
}

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('Liftoff!');
} else {
   console.log('Launch scrubbed.');
}
```

Run it at repl.it

Should the shuttle have launched? Did it?

Check your answer.

2. Let's break the code down into smaller chunks. Consider the first `if/else` block below. We've commented out some of the variables we're not inspecting right now. Add `console.log(launchReady)` after this block, then run the program.

```
let launchReady = false;
let fuelLevel = 17000;
// let crewStatus = true;
// let computerStatus = 'green';

if (fuelLevel >= 20000) {
   console.log('Fuel level cleared.');
   launchReady = true;
} else {
   console.log('WARNING: Insufficient fuel!');
   launchReady = false;
}
```

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. Here again, we comment the variables and blocks that we're not inspecting. Add another `console.log(launchReady)` after this block and run the program.

```
let launchReady = false;
// let fuelLevel = 17000;
let crewStatus = true;
let computerStatus = 'green';

// if (fuelLevel >= 20000) {
//    console.log('Fuel level cleared.');
//    launchReady = true;
// } else {
//    console.log('WARNING: Insufficient fuel!');
//    launchReady = false;
// }

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

Run it at repl.it

Given `crewStatus` and `computerStatus`, should `launchReady` be `true` or `false` after this check?

Check your answer.

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

```
let launchReady = false;
let fuelLevel = 17000;
let crewStatus = true;
let computerStatus = 'green';

if (fuelLevel >= 20000) {
   console.log('Fuel level cleared.');
   launchReady = true;
} else {
   console.log('WARNING: Insufficient fuel!');
   launchReady = false;
}
```

```
13 console.log(launchReady);
14
15 if (crewStatus && computerStatus === 'green'){
16    console.log('Crew & computer cleared.');
17    launchReady = true;
18 } else {
19    console.log('WARNING: Crew or computer not ready!');
20    launchReady = false;
21 }
22 console.log(launchReady);
```

Run it at repl.it

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.

   The issue is with the `launchReady` value being assigned and reassigned based on different checks. One way to fix the logic error is to use two different variables to store the results of checking the fuel readiness (lines 6-13) and checking the crew and computer readiness (lines 15-22).

   Update your code to do this. Verify that your change works by updating the `console.log` statements.

   Fix it at repl.it

   Check your solution.

6. Almost done, so wipe the sweat off your 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.

- ← 6.7. Asking Good Questions
- 7. Stringing Characters Together →

Back to top

LaunchCode logo

Search

1. Contents
2. 7. Stringing Characters Together

7. Stringing Characters Together¶

- ← 6.8. Exercises: Debugging
- 7.1. Strings as Collections →

Back to top

LaunchCode logo

Search

1. Contents
2. 7. Stringing Characters Together
3. 7.1. Strings as Collections

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.

The string "JavaScript" broken down into individual letters.

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. Stringing Characters Together
- 7.2. Bracket Notation →

Back to top

LaunchCode logo

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.

The string "JavaScript" with indices labeled below each letter

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.

```
let jsCreator = "Brendan Eich";
let firstInitial = jsCreator[0];
let lastInitial = jsCreator[8];

let outputStr = "JavaScript was created by somebody with initials " +
   firstInitial + "." +
   lastInitial + ".";

console.log(outputStr);
```

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!

```
let jsCreator = "Brendan Eich";

console.log(jsCreator[-1]);
console.log(jsCreator[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?

```
let phrase = "JavaScript rocks!";
console.log(phrase[phrase.length - 8]);
```

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

- ← 7.1. Strings as Collections
- 7.3. Strings as Objects →

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.

```
console.log(typeof console);
```

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 firstName = "Grace";
2 let lastName = "Hopper";
3
4 console.log(firstName, "has", firstName.length, "characters");
5 console.log(lastName, "has", lastName.length, "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

```
let nonprofit = "LaunchCode";

console.log(nonprofit.toLowerCase());
console.log(nonprofit);
```

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`?

Question

What is the length of `location`?

```
cityName = "Vienna";
stateName = "Virginia";
location = cityName + ", " +  stateName;

console.log(location.length);
```

1. 16
2. 17
3. 15
4. 14

- ← 7.2. Bracket Notation
- 7.4. String Immutability →

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 nonprofit = "Launchcode";
2
3 console.log(nonprofit);
4 nonprofit[6] = "C";
5 console.log(nonprofit);
```

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 nonprofit = "Launchcode";
2 nonprofit = "LaunchCode";
3
4 console.log(nonprofit);
```

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:

A variable, nonprofit, pointing at "LaunchCode" with a lowercase-c.

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?

- ← 7.3. Strings as Objects
- 7.5. String Methods →

Back to top

LaunchCode logo

Search

1. Contents
2. 7. Stringing Characters Together
3. 7.5. String Methods

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

```
let nonprofit = "LaunchCode";

console.log(nonprofit.toLowerCase());
console.log(nonprofit);
```

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

Common String Methods¶

Method	Syntax	Description
indexOf	`stringName.indexOf(substr)`	Returns the index of the first occurrence of the substring in the string, and returns -1 if the substring is not found.
toLowerCase	`stringName.toLowerCase()`	Returns a copy of the given string, with all uppercase letters converted to lowercase.
toUpperCase	`stringName.toUpperCase()`	Returns a copy of the given string, with all lowercase letters converted to uppercase.
trim	`stringName.trim()`	Returns a copy of the given string with the leading and trailing whitespace removed.

Method	Syntax	Description
replace	`stringName.replace(searchChar, replacementChar)`	Returns a copy of `stringName`, with the first occurrence of `searchChar` replaced by `replacementChar`.
slice	`stringName.slice(i, j)`	Returns the substring consisting of characters from index `i` through index `j-1`.

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 language = "JavaScript";
2 language.replace('J', 'Q');
3 language.slice(0,5);
4 console.log(language);
```

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?

```
let org = "  The LaunchCode Foundation ";
let trimmed = org.trim();

console.log(trimmed);
```

1. `"  The LaunchCode Foundation "`
2. `"The LaunchCode Foundation"`
3. `"TheLaunchCodeFoundation"`
4. `" The LaunchCode Foundation"`

- ← 7.4. String Immutability
- 7.6. Encoding Characters →

Back to top

LaunchCode logo

Search

1. Contents
2. 7. Stringing Characters Together
3. 7.6. Encoding Characters

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.

7.6.2.1. 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 🌥.

7.6.2.2. 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

```
let nonprofit = "LaunchCode";

console.log(nonprofit.charCodeAt(0));
console.log(nonprofit.charCodeAt(1));
console.log(nonprofit.charCodeAt(2));
console.log(nonprofit.charCodeAt(3));
console.log(nonprofit.charCodeAt(4));
console.log(nonprofit.charCodeAt(5));
console.log(nonprofit.charCodeAt(6));
console.log(nonprofit.charCodeAt(7));
console.log(nonprofit.charCodeAt(8));
console.log(nonprofit.charCodeAt(9));
```

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

```
let codes = [76, 97, 117, 110, 99, 104, 67, 111, 100, 101];

let characters = String.fromCharCode(codes[0]) + String.fromCharCode(cod
               + String.fromCharCode(codes[2]) + String.fromCharCode(cod
               + String.fromCharCode(codes[4]) + String.fromCharCode(cod
               + String.fromCharCode(codes[6]) + String.fromCharCode(cod
               + String.fromCharCode(codes[8]) + String.fromCharCode(cod

console.log(characters);
```

Console Output

```
LaunchCode
```


- ← 7.5. String Methods
- 7.7. Special Characters →

Back to top

LaunchCode logo

Search

1. Contents
2. 7. Stringing Characters Together
3. 7.7. Special Characters

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 a \ (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

```
console.log("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

```
console.log("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

```
console.log("\"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 prints `'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');`

- ← 7.6. Encoding Characters
- 7.8. Template Literals →

Back to top

LaunchCode logo

7.8. Template Literals¶

Earlier, we used *concatenation* to combine strings and variables together in order to create specific output:

Example

```
let name = "Jack";
let currentAge = 9;

console.log("Next year, " + name + " will be " + (currentAge + 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.

```
let name = "Jack";
let currentAge = 9;

console.log(`Next year, ${name} will be ${currentAge + 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 poem = `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.log(poem);
```

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 pluralNoun = ;
2 let name = ;
```

```
3 let verb = ;
4 let adjective = ;
5 let color = ;
6
7 console.log("JavaScript provides a "+ color +" collection of tools – inc
```

repl.it

- ← 7.7. Special Characters
- 7.9. Exercises: Strings →

Back to top

LaunchCode logo

Search

1. Contents
2. 7. Stringing Characters Together
3. 7.9. Exercises: Strings

7.9. Exercises: Strings¶

7.9.1. Part One¶

1. 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).
        1. Modify your code to print out the length of a decimal value EXCLUDING the period.
    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

    Check your solution.

7.9.2. Part Two¶

1. 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

2. Let's use string methods to do more work on the DNA strand:

    1. Replace the sequence `'GCT'` with `'AGG'`, and then print the altered strand.
    2. Look for the sequence `'CAT'` with `indexOf()`. If found print, `'CAT found'`, otherwise print, `'CAT NOT found'`.
    3. Use `slice()` to print out the fifth set of 3 characters (called a **codon**) from the DNA strand.
    4. Use a template literal to print, `"The DNA strand 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

    Check your solution.

7.9.3. Part Three¶

1. 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()` methods 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

2. 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

Check your solution.

- ← 7.8. Template Literals
- 8. Arrays Keep Things in Order →

Back to top

LaunchCode logo

Search

1. Contents
2. 8. Arrays Keep Things in Order

8. Arrays Keep Things in Order¶

- ← 7.9. Exercises: Strings
- 8.1. Arrays Are Like Strings →

Back to top

LaunchCode logo

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 of named languages. The array contains four strings, each of those values has an index position.

A label, languages, pointing to an array that contains "Python" at index 0, "C#" at index 1, "Java" at index 2, and "JavaScript" at index 3.

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.

```
let emptyArray = [];

let programmingLanguages = ["JavaScript", "Python", "Java", "C#"];
```

Array items can also be declared on multiple lines.

```
let javaScriptFrameworks = [
   "React",
   "Angular",
   "Ember",
   "Vue"
];
```

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.

```
let emptyArray = [];
console.log(emptyArray.length);

let programmingLanguages = ["JavaScript", "Python", "Java", "C#"];
console.log(programmingLanguages.length);
```

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 grabBag = ["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.

```
let classes = ["science, computer, art"];

let teachers = ["Jones", "Willoughby", "Rhodes"];
```

How can you change the `classes` array declaration to have the same number of items as the `teachers` array?

- ← 8. Arrays Keep Things in Order
- 8.2. Working With Arrays →

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.

```
let programmingLanguages = [
   "JavaScript", // index 0
   "Python",     // index 1
   "Java",       // index 2
   "C#"          // index 3
];
console.log(programmingLanguages[0]);
console.log(programmingLanguages[3]);

// What will happen when index 4 is requested?
console.log(programmingLanguages[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.

```
let programmingLanguages = ["JavaScript", "Python", "Java", "C#"];
console.log(programmingLanguages[-1]);
console.log(programmingLanguages[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.

```
let javaScriptFrameworks = ["React", "Angular", "Ember"];
console.log(javaScriptFrameworks);

// Set the value of index 2 to be "Vue"
javaScriptFrameworks[2] = "Vue";

// Notice the value at index 2 is now "Vue"
console.log(javaScriptFrameworks);
```

Console Output

```
[ 'React', 'Angular', 'Ember' ]
[ 'React', 'Angular', 'Vue' ]
```


- ← 8.1. Arrays Are Like Strings
- 8.3. Array Methods →

Back to top

LaunchCode logo

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.

Methods That Return Information About The Array¶

Method	Syntax	Description
includes	`arrayName.includes(item)`	Checks if an array contains the specified item.
indexOf	`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.

Methods That Rearrange The Entries In The Array¶

Method	Syntax	Description
reverse	`arrayName.reverse()`	Reverses the order of the elements in an array.
sort	`arrayName.sort()`	Arranges the elements of an array into increasing order (kinda).

Methods That Add Or Remove Entries From An Array¶

Method	Syntax	Description
pop	`arrayName.pop()`	Removes and returns the LAST element in an array.
push	`arrayName.push(item1, item2, ...)`	Adds one or more items to the END of an array and returns the new length.
shift	`arrayName.shift()`	Removes and returns the FIRST element in an array.
splice		

Method	Syntax	Description
	`arrayName.splice(index, number, item1, item2, ...)`	Adds, removes or replaces one or more elements anywhere in the array.
unshift	`arrayName.unshift(item1, item2, ...)`	Adds one or more items to the START of an array and returns the new length.

Methods That Create New Arrays¶

Method	Syntax	Description
concat	`arr.concat(otherArray1, otherArray2, ...)`	Combines two or more arrays and returns the result as a new array.
join	`arr.join('connecter')`	Combines all the elements of an array into a string.
slice	`arr.slice(start index, end index)`	Copies selected entries of an array into a new array.
split	`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?

```
let charles = ['coder', 'Tech', 47, 23, 350];
charles.sort();
console.log(charles);
```

1. `[350, 23, 47, 'Tech', 'coder']`
2. `['coder', 'Tech', 23, 47, 350]`
3. `[23, 47, 350, 'coder', 'Tech']`
4. `[23, 350, 47, 'Tech', 'coder']`

Question

Which statement converts the string `str = 'LaunchCode students rock!'` into the array `['LaunchCode', 'students', 'rock!']`?

1. `str.join(" ");`
2. `str.split(" ");`
3. `str.join("");`
4. `str.split("");`

Question

What is printed by the following program?

```
let groceryBag = ['bananas', 'apples', 'edamame', 'chips', 'cucumbers',
let selectedItems = [];

selectedItems = groceryBag.slice(2, 5).sort();
console.log(selectedItems);
```

1. ['chips', 'cucumbers', 'edamame']
2. ['chips', 'cucumbers', 'edamame', 'milk']
3. ['cheese', 'chips', 'cucumbers']
4. ['cheese', 'chips', 'cucumbers', 'edamame']

- ← 8.2. Working With Arrays
- 8.4. Multi-Dimensional Arrays →

Back to top

LaunchCode logo

Search

1. Contents
2. 8. Arrays Keep Things in Order
3. 8.4. Multi-Dimensional Arrays

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.

A label, synonyms, pointing to an array that contains arrays at it's four indexes. Each inner array has a list of words that are synonyms.

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".

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.

```
let shuttleCrews = [
   ['Robert Gibson', 'Mark Lee', 'Mae Jemison'],
   ['Kent Rominger', 'Ellen Ochoa', 'Bernard Harris'],
   ['Eilen Collins', 'Winston Scott',  'Catherin Coleman']
];

console.log(shuttleCrews[0][2]);
console.log(shuttleCrews[1][1]);
console.log(shuttleCrews[2][1]);
```

Console Output

```
Mae Jemison
Ellen Ochoa
Winston Scott
```

8.4.2. Multi-Dimensions and Array Methods¶

In a multi-dimensional array, both the inner and outer arrays can be altered with array methods. However, bracket notation must be used correctly.

To apply a method to the outer array, the syntax is:

```
multiArrayName.method();
```

To apply a method to one of the inner arrays, the syntax is:

```
multiArrayName[indexOfInnerArray].method();
```

Example

Use array methods to add an additional crew array and alter existing arrays.

```
let shuttleCrews = [
   ['Robert Gibson', 'Mark Lee', 'Mae Jemison'],
```

```
  ['Kent Rominger', 'Ellen Ochoa', 'Bernard Harris'],
  ['Eilen Collins', 'Winston Scott',  'Catherin Coleman']
];

let newCrew = ['Mark Polansky', 'Robert Curbeam', 'Joan Higginbotham'];

// Add a new crew array to the end of shuttleCrews
shuttleCrews.push(newCrew);
console.log(shuttleCrews[3][2]);

// Reverse the order of the crew at index 1
shuttleCrews[1].reverse();
console.log(shuttleCrews[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"`?

```
let school = [
   ["science", "computer", "art"],
   ["Jones", "Willoughby", "Rhodes"]
];
```

How would you add `"dance"` to the array at `school[0]`?

How would you add `"Holmes"` to the array at `school[1]`?

- ← 8.3. Array Methods
- 8.5. Exercises: Arrays →

Back to top

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.

    Check your solution.

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 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 ``()``?

Check your solution.

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

    Check your solution.

    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

    ```
    holdCabinet1 ['duct tape', 'gum', 3.14, false, 6.022e23]
    ```

    and

    ```
    holdCabinet2 ['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.

    Check your solution.

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.

    Check your solution.

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:
        1. `element1 = ['hydrogen', 'H', 1.008]`
        2. `element2 = ['helium', 'He', 4.003]`
        3. `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

    Check your solution.

Excellent work, records keeper. Welcome aboard.

- ← 8.4. Multi-Dimensional Arrays
- 8.6. Studio: Strings and Arrays →

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.

1. 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.
2. Modify your code to accept user input. Query the user to enter the number of letters that will be relocated.
3. 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¶

Note

The starter code for this section contains unit tests. You will see a lot of new code in the starter code. The directions will tell you which function to work in. Look for the `TODO` and `NOTE` comments for guidance and direction. The Test Code appendix page provides more information and tips for working inside code like this.

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.

1. For a given string, use the `includes` method to check to see if the words are separated by commas (`,`), semicolons (`;`), or just spaces.
2. Use the `reverseCommas()` function to code the following. 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. For example, `"up,to,code,fun"` becomes `"fun,code,to,up"`.
3. Use the `semiDash()` function to code the following. 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 hyphen-separated string. For example, `"up;to;code;fun"` becomes `"code-fun-to-up"`.
4. Use the `reverseSpace()` function to code the following. 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. For example, `"to code up fun"` becomes `"up to fun code"`.
5. Use the `commaSpace()` function to code the following. *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!

1. 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.
    1. `"water bottles, meal packs, snacks, chocolate"`
    2. `"space suits, jet packs, tool belts, thermal detonators"`
    3. `"parrots, cats, moose, alien eggs"`
    4. `"blankets, pillows, eyepatches, alarm clocks"`
2. Initialize a `cargoHold` array and add the cabinet arrays to it. Print `cargoHold` to verify its structure.
3. Query the user to select a cabinet (0-3) in the `cargoHold`.
4. 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.
5. *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 ____."`

9. Repeating With Loops¶

Back to top

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 console.log(0);
2 console.log(1);
3 console.log(2);
4 console.log(3);
5 console.log(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 i = 0; i < 51; i++) {
2   console.log(i);
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. Repeating With Loops
- 9.2. `for` Loops →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.2. `for` Loops

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 i = 0; i < 51; i++) {
2    console.log(i);
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.
    1. If the condition is `true` then the loop executes again.
    2. 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 i = 0; i < 51; i++) {
2    console.log(i);
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.

../../_images/for-loop-flow.png

Flow of execution of a `for` loop¶

- ← 9.1. Iteration
- 9.3. Iterating Over Collections →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.3. Iterating Over Collections

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

```
let name = "LaunchCode";

for (let i = 0; i < name.length; i++) {
```

```
4    console.log(name[i]);
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.

```
1 // create a string variable containing your name
2
3 // write a for loop that prints each character in your name on a differe
```

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; i < languages.length; i++) {
4    console.log(languages[i]);
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

- ← 9.2. `for` Loops
- 9.4. Breaking Down the `for` Statement →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.4. Breaking Down the `for` Statement

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; i < 51; i++) {
2   console.log(i);
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 i = 0; i < upperBound; i++) {
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.

9.4.1.1. 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 i = 3; i < 10; i++) {
2    console.log(i);
3 }
```

This loop prints each of the letters `C`, `o`, `d`, and `e` on a separate line.

```
1 let name = "LaunchCode";
2
3 for (let i = 6; i < name.length; i++) {
4    console.log(name[i]);
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 i = 0; i < 42; i++) {
2   console.log("LaunchCode");
3 }
```

9.4.1.2. 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 i = 0; i < -1; i++) {
2   console.log("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 i = 0; i > -1; i++) {
2   console.log("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.

9.4.1.3. 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 i = 0; i < 51; i = i + 2) {
2   console.log(i);
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 i = 0; i < 51; i--) {
2   console.log(i);
3 }
```

Try It!

How does each of the three components affect the behavior of a `for` loop? Experiment by modifying each of them in this example: the variable initialization, the boolean condition, and the update expression.

```
1 for (let i = 0; i < 51; i++) {
2   console.log(i);
3 }
```

repl.it

9.4.2. Check Your Understanding¶

Consider the program:

```
let phrase = "LaunchCode's LC101";

for (let i = 0; i < phrase.length - 1; i = i + 3) {
   console.log(phrase[i]);
}
```

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 '`

- ← 9.3. Iterating Over Collections
- 9.5. The Accumulator Pattern →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.5. The Accumulator Pattern

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

```
let n = 6;
let total = 0;

for (let i = 1; i <= n; i++) {
   total += i;
}

console.log(total);
```

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.

```
let str = "blue";
let reversed = "";
```

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

```
let str = "blue";
let reversed = "";

for (let i = 0; i < str.length; i++) {
   reversed = str[i] + reversed;
}

console.log(reversed);
```

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.

The accumulator pattern, step by step¶

Loop iteration	i	str[i]	reversed
(before first iteration)	not defined	not defined	""
1	0	"b"	"b"
2	1	"l"	"lb"

Loop iteration	i	str[i]	reversed
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...n as we did above, with the difference being we will use the items in an array rather than 1...n.

Example

```
1 let numbers = [2, -5, 13, 42];
2 let total = 0;
3
4 for (let i = 0; i < numbers.length; i++) {
5    total += numbers[i];
6 }
```

Console Output

```
52
```


- ← 9.4. Breaking Down the `for` Statement
- 9.6. `while` Loops →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.6. `while` Loops

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.

../../_images/while-loop-flow.png

Flow of execution of a `while` loop¶

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.

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 i = 0; i < 51; i++) {
2    console.log(i);
3 }
```

This can be rewritten as a while loop:

```
1 let i = 0;
2
3 while (i < 51) {
4    console.log(i);
```

```
5    i++;
6 }
```

repl.it

Instead of relying on 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 input = require('readline-sync');
2
3 let num = input.question('Please enter a positive number:');
4 num = Number(num);
5
6 while (num <= 0) {
7    num = input.question('Invalid input. Please enter a positive number:'
8    num = Number(num);
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.

```
let i = 0;

while (i < 51) {
   console.log(i);
}
```

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 never-ending list of zeros.

Even when we remember to update the counter, we must be careful to make sure that the condition will eventually be `false`.

```
let i = 0;

while (i < 51) {
   console.log(i);
   i--;
}
```

In this case, `i--` decreases the value of the counter. Since `i` starts at 0, `i < 51` will always be true. If you ran this program, you would see an ever-expanding list of negative numbers.

Tip

At some point, everyone creates an infinite loop. When this happens to you, typing `control-c` will usually force your program to stop.

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?

```
let n = 10;
let answer = 1;

while (n > 0) {
   answer = answer + n;
   n = n + 1;
}

console.log(answer);
```

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.

- ← 9.5. The Accumulator Pattern
- 9.7. Terminating a Loop With `break` →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.7. Terminating a Loop With `break`

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.

```
for (let i = 0; i < 42; i++) {

   // rest of loop body

   if (i > 10) {
      break;
   }

}
```

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.

```
let numbers = [ /* some numbers */ ];
let searchVal = 42;
let i = 0;

while (i < numbers.length) {
   if (numbers[i] === searchVal) {
      break;
   }
   i++;
}

if (i < numbers.length) {
   console.log("The value", searchVal, "was located at index", i);
} else {
   console.log("The value", searchVal, "is not in the array.");
}
```

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.

- ← 9.6. `while` Loops
- 9.8. Choosing Which Loop to Use →

Back to top

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

- ← 9.7. Terminating a Loop With `break`
- 9.9. Exercises: Loops →

Back to top

LaunchCode logo

Search

1. Contents
2. 9. Repeating With Loops
3. 9.9. Exercises: Loops

9.9. Exercises: Loops¶

> Practice makes better. Repetition is a good thing.
>
> Repetition is a good thing.
>
> Repetition is a good thing.
>
> 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 i = 0; i < 100; i++){
2     console.log("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:
    1. Print the numbers 0 - 20, one number per line.
    2. Print only the ODD values from 3 - 29, one number per line.
    3. Print the EVEN numbers 12 down to -14 in descending order, one number per line.
    4. 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:

    1. Print each element of the array to a new line.
    2. 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:

    1. Define an empty `evens` array to hold the even numbers and an `odds` array for the odd numbers.
    2. In the loop, determine if each number is even or odd, then put that number into `evens` or `odds`, as appropriate.
    3. Print the arrays to confirm the results. Print `evens` first. Example: `console.log(evens);`

    Check your solution.

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.

1. Construct `while` loops to do the following:

    1. 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.
    2. 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.
    3. 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. (Hint: The loop should end when there is not enough fuel to boost the crew another 50 km, so the fuel level might not reach 0).

2. 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."

        Check your solution.

- ← 9.8. Choosing Which Loop to Use
- 9.10. Studio: Loops →

9.10. Studio: Loops¶

Now that we've launched our shuttle, let's use loops (iteration) to automate some tasks.

Fork this repl to start coding. Before you dive in, you might notice that we have several files and folders inside. Your work will go into `solution.js`, but please feel free to explore the program and please don't edit anything outside `solution.js`.

9.10.1. Part A: Put dinner together¶

1. First, inside `solution.js`, use the variables provided to store the following arrays.
    - 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'
      ```

1. Inside of `mealAssembly()`, write a `for` loop to assemble `numMeals` meals.
    1. The meals must include one item from each category in the `pantry` array.

Hint

The computer needs to know how many crew members to prepare food for and what ingredients. Consider creating a nested loop that will create a meal for each crew member and then add it into a larger collection of meals.

2. Each ingredient can only be used ONCE.
3. Add each meal to the `meals` array once it is assembled.
4. To test your own solution, scroll down to another function defined in this file called `runProgram()`. Uncomment the lines that call the `mealAssembly` function and print the result. Use the repl.it 'Run' button as usual to see your results printed to the console.

9.10.2. Part B: Collect User Input¶

Update `askForNumber()` to add user input and validation.

1. 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.
2. Save the result to the `numMeals` variable returned by the function.
3. Test your solution to this part by returning to the `runProgram()` function in the file and uncommenting the section labelled for testing Part B.

9.10.3. Checking Your Work¶

If you want to make sure that you have checked all the boxes, run the following command in your shell on replit.

```
npm test
```

This command runs the Jasmine tests that are checking your work for you. If you have a test that fails, check out the name of the test to get a hint as to what you are missing. If you need a refresher on how running the tests works, check out the appendix on Tested Code.

9.10.4. Bonus Mission¶

Working and living aboard this amazing space shuttle requires you to pay the utmost attention to cybersecurity. Once you are done working on the meal system, you are prompted to create a new password that will be used for the next 24 hours. Having run out of strong password ideas, your shuttle captain has encouraged you to make a password generator for yourself.

Write your code inside `generatePassword()`.

1. Construct a `for` loop that combines the two strings together, alternating the characters from each source, and saves the combined string to the `code` variable.

   Examples

   1. If `string1 = "1234"` and `string2 = "5678"`, then the output will be "15263748".
   2. If `string1 = "ABCDEF"` and `string2 = "notyet"`, then the output will be "AnBoCtDyEeFt".
   3. If `string1 = "LoOt"` and `string2 = "oku!"`, then the output will be "LookOut!".

- ← 9.9. Exercises: Loops
- 10. Functions Are at Your Beck and Call →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call

10. Functions Are at Your Beck and Call¶

Back to top

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).

```
console.log("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

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 = Number("42");
2 console.log("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.

```
let commaSeparatedValues = "Smith,Jane,100 Cherry Blossom Lane";
let values = commaSeparatedValues.split(',');
console.log(values);
```

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.

A "function machine," consisting of a box which takes inputs, and from which output emerges.

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) = x² + 4x - 2
```

We can *call* the function by giving it a specific *input*:

```
f(3) = 3² + 4*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. Functions Are at Your Beck and Call
- 10.2. Using Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.2. Using Functions

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 stringVal = String(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.

```
console.log(String(42));
```

Console Output

```
42
```

If a function doesn't provide an explicit return value, the special value `undefined` will be returned.

Example

```
1 let returnVal = console.log("LaunchCode");
2 console.log(returnVal);
```

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.

- ← 10.1. Introduction
- 10.3. Creating Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.3. Creating Functions

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:

```
function myFunction(parameter1, parameter2,..., parameterN) {

   // function body

}
```

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.

10.3.1.1. An Example¶

Let's see function syntax in action. We first consider a program that prints an array of names.

```
let names = ["Lena", "James", "Julio"];

for (let i = 0; i < names.length; i++) {
   console.log(names[i]);
}
```

Following this pattern, we can create a function that prints *any* array of names.

```
function printNames(names) {
   for (let i = 0; i < names.length; i++) {
      console.log(names[i]);
   }
}
```

Breaking down the components of a function using our new terminology gives us:

- **Function name**: `printNames`
- **Parameter(s)**: `names`
- **Body**:

```
for (let i = 0; i < names.length; i++) {
   console.log(names[i]);
}
```

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.

```
function printArray(names) {
   for (let i = 0; i < names.length; i++) {
      console.log(names[i]);
   }
}

```

```
7 printArray(["Lena", "James", "Julio"]);
8 console.log("---");
9 printArray(["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 sayHello() {
2    console.log("Hello, World!");
3 }
```

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

```
3 }
4
5 sayHello();
```

Console Output

```
Hello, World!
```


- ← 10.2. Using Functions
- 10.4. Function Input and Output →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.4. Function Input and Output

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`.

10.4.1.1. 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 someVal;
```

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 sumToN(n) {
2    let sum = 0;
```

```
3    for (let i = 0; i <= n; i++) {
4       sum += i;
5    }
6    return sum;
7 }
8
9 console.log(sumToN(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?

10.4.1.2. 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 doNothing() {}
```

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 doNothing() {}
2
3 let returnVal = doNothing();
4 console.log(returnVal);
```

Console Output

```
undefined
```

10.4.1.3. `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 pastThePointOfReturn() {
2    return "I'm done!";
3    console.log("This will not be printed");
4 }
5
6 console.log(pastThePointOfReturn());
```

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 countToN(n) {
 2    let count = 1;
 3    while (true) {
 4       if (count > n) {
 5          return;
 6       }
 7       console.log(count);
 8       count++;
 9    }
10 }
```

10.4.1.4. 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 isEven(n) {
 2    if (n % 2 === 0) {
 3       return true;
 4    } else {
 5       return false;
 6    }
 7 }
 8
 9 console.log(isEven(4));
10 console.log(isEven(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 isEven(n) {
2    if (n % 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 isEven(n) {
2    return n % 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 hello(name) {
2    return `Hello, ${name}!`;
3 }
4
5 console.log(hello("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

```
function hello(name) {
   return `Hello, ${name}!`;
}

console.log(hello());
```

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.

```
function hello(name = "World") {
   return `Hello, ${name}!`;
}

console.log(hello());
console.log(hello("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.

```
// returns "Launch"
"LaunchCode".slice(0, 6);

// returns "Code"
"LaunchCode".slice(6);

// also returns "Code"
"LaunchCode".slice(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.

```
function hello(name = "World") {
   return `Hello, ${name}!`;
}

console.log(hello("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?

```
function plusTwo(num) {
   return num + 2;
}

let a = 2;

```

```
for (let i=0; i < 4; i++) {
   a = plusTwo(a);
}

console.log(a);
```

Question

What does the following function *return*?

```
function repeater(str) {
   let repeated = str + str;
   console.log(repeated);
}

repeater('Bob');
```

1. `"BobBob"`
2. Nothing (no return value)
3. `undefined`
4. The value of `Bob`

Question

What does the following code *output*?

```
function repeater(str) {
   let repeated = str + str;
   console.log(repeated);
}

repeater('Bob');
```

1. `"BobBob"`
2. Nothing (no output)
3. `undefined`
4. The value of `Bob`

- ← 10.3. Creating Functions
- 10.5. A Good Function-Writing Process →

Back to top

LaunchCode logo

Search

1. Contents

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:

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 makeSandwich(breadType, fillingType, condiments) {
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.

- ← 10.4. Function Input and Output
- 10.6. Parameters and Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.6. Parameters and Variables

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 removeHyphens(str) {
 2    let strWithoutHyphens = ''
 3
 4    for (let i = 0; i < str.length; i++) {
 5       if (str[i] !== '-') {
 6          strWithoutHyphens += str[i];
 7       }
 8    }
 9
10    return strWithoutHyphens;
11 }
12
13 let launchCodePhone = "314-254-0107";
14 console.log(removeHyphens(launchCodePhone));
15 console.log(strWithoutHyphens);
```

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.

```
let message = "Hello, World!";

function printMessage() {
   console.log(message);
}

printMessage();
```

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.

There is, however, a specific type of variable that is visible to every function, known as a **global variable**. We remarked earlier that a global variable is

created when initializing a variable without using `let`, `const`, or `var`, and they should be used very rarely.

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 message = "Hello, World!";
2
3 function printMessage() {
4    console.log(message);
5 }
6
7 printMessage();
8 message = "Goodbye";
9 printMessage();
```

repl.it

An interesting thing happens when a function parameter has the same name as a variable that is in-scope.

Example

```
1 let message = "Hello, World!";
2
3 function printMessage(message) {
4    console.log(message);
5 }
6
7 printMessage("Goodbye");
```

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 called **shadowing**, based on the metaphor that a function parameter "casts it's 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?

```
let num = 42;

function isEven (num) {
   return num % 2 === 0;
}

console.log(isEven(43));
```


- ← 10.5. A Good Function-Writing Process
- 10.7. Naming Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.7. Naming Functions

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-resistent.

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 pair can go a long way toward making it clear what a function 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

- `convertCelsiusToFahrenheit`
- `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?

```
function myFunc(radius) {
  let area = Math.PI * radius**2;
  return area;
}
```

1. `area`
2. `calculateAreaOfCircle`
3. `circle`
4. `shape`

- ← 10.6. Parameters and Variables
- 10.8. Composing Functions →

Back to top

LaunchCode logo

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 reversed a string, wouldn't it?

10.8.1.1. The `reverse` Function¶

Let's write a function that, given a string, returns its reverse.

One approach uses the accumulator pattern:

```
1 function reverse(str) {
2    let reversed = '';
3
4    for (let i = 0; i < str.length; i++) {
5       reversed = str[i] + reversed;
6    }
7
```

```
8    return reversed;
9 }
```

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 reverse(str) {
2    let lettersArray = str.split('');
3    let reversedLettersArray = lettersArray.reverse();
4    return reversedLettersArray.join('');
5 }
```

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.

10.8.1.2. 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 reverse(str) {
2    return str.split('').reverse().join('');
3 }
4
5 function isPalindrome(str) {
6    return reverse(str) === str;
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 isPalindrome(str) {
2    let reversed = str.split('').reverse().join('');
3    return reversed === str;
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 makeSandwich( /*parameters*/ ) {
 2    // make the sandwich
 3 }
 4
 5 function pourDrink( /*parameters*/ ) {
 6    // pour the drink
 7 }
 8
 9 function makeLunch( /*parameters*/ ) {
10    makeSandwich( /*parameters*/ );
```

```
11    pourDrink( /*parameters*/ );
12 }
```

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.

- ← 10.7. Naming Functions
- 10.9. Why Create Functions? →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.9. Why Create Functions?

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 isPalindrome(str) {
 2
 3    let reversed = '';
 4
 5    for (let i = 0; i < str.length; i++) {
 6       reversed = str[i] + reversed;
 7    }
 8
 9    return reversed === str;
10 }
```

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 reverse(str) {
 2    let reversed = '';
 3
 4    for (let i = 0; i < str.length; i++) {
 5       reversed = str[i] + reversed;
 6    }
 7    return reversed;
 8 }
 9
10 function isPalindrome(str) {
11    return reverse(str) === str;
12 }
```

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.

- ← 10.8. Composing Functions
- 10.10. Exercises: Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 10. Functions Are at Your Beck and Call
3. 10.10. Exercises: Functions

10.10. Exercises: Functions¶

To solve 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 solve problems with code, you need to be skilled at writing functions. And to master functions, you need to write a *lot* of them.

These exercises ask you to write many 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 `makeLine(size)` that returns a line with exactly `size` hashes.

   ```
   console.log(makeLine(5));
   ```

   Console Output

   ```
   #####
   ```

   Check your solution.

2. Write a function called `makeSquare(size)` that returns a `size` by `size` string of hashes. The function should NOT print each row of the square. Instead, it must return a single string that contains the entire shape.

   Tip

   1. Call your `makeLine` function to create each row of the square.
   2. The newline character, `\n`, will be helpful to you.
   3. Do NOT include a newline character at the end of your string.

   ```
   console.log(makeSquare(5));
   ```

   Console Output

   ```
   #####
   #####
   #####
   #####
   #####
   ```

   Warning

   For each of the shape exercises, do not include a newline character at the very end of your string. While the final `\n` might not be noticeable for the simpler shapes, including it will make life harder for you toward the end of the exercises.

3. Write a function `makeRectangle(width, height)` that returns a rectangle with the given width and height. Use your `makeLine` function to do this.

   ```
   console.log(makeRectangle(5, 3));
   ```

   Console Output

   ```
   #####
   #####
   #####
   ```

   Check your solution.

4. Now, go back and rewrite `makeSquare` to use `makeRectangle`.

10.10.2. Triangles¶

1. Write a function `makeDownwardStairs(height)` that prints the staircase pattern shown below, with the given height. Use your `makeLine` function to do this.

   ```
   console.log(makeDownwardStairs(5));
   ```

   Console Output

   ```
   #
   ##
   ###
   ####
   #####
   ```

   Check your solution.

2. Write a function `makeSpaceLine(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.

   ```
   console.log(makeSpaceLine(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 `makeIsoscelesTriangle(height)` that returns a triangle of the given height.

   ```
   console.log(makeIsoscelesTriangle(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.

Check your solution.

10.10.3. Diamonds¶

1. Write a function `makeDiamond(height)` that returns a diamond where the triangle formed by the *top* portion has the given height.

   ```
   console.log(makeDiamond(5));
   ```

   Console Output

   ```
       #
      ###
     #####
    #######
   #########
   #########
    #######
     #####
      ###
       #
   ```

   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 `'#'`.

- ← 10.9. Why Create Functions?
- 10.11. Studio: Functions →

Back to top

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

```
let arr = ['hello', 'world', 123, 'orange'];

arr.reverse()
console.log(arr);
```

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.
    1. Define the function as `reverseCharacters`. Give it one parameter, which will be the string to reverse.
    2. Within the function, `split` the string into an array, then reverse the array.
    3. Use `join` to create the reversed string and *return* that string from the function.
    4. Below the function, define and initialize a variable to hold a string.

5. Use `console.log(reverseCharacters(myVariableName));` to call the function and verify that it correctly reverses the characters in the string.
6. *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:

1. `'apple'`
2. `'LC101'`
3. `'Capitalized Letters'`
4. `'I love the smell of code in the morning.'`

10.11.2. Reverse Digits¶

1. 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"`).
    1. Add an `if` statement to `reverseCharacters` to check the `typeof` the parameter.
    2. If `typeof` is 'string', return the reversed string as before.
    3. If `typeof` is 'number', convert the parameter to a string, reverse the characters, then convert it back into a number.
    4. Return the reversed number.
    5. 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:

1. `1234`
2. `'LC101'`
3. `8675309`
4. `'radar'`

10.11.3. Complete Reversal¶

1. 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:
    1. Define and initialize an empty array.
    2. Loop through the old array.

3. For each element in the old array, call `reverseCharacters` to flip the characters or digits.
4. Add the reversed string (or number) to the array defined in part 'a'.
5. Return the final, reversed array.
6. *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**
`['apple', 'potato', 'Capitalized Words']`	`['sdroW dezilatipaC', 'otatop', 'elppa']`
`[123, 8897, 42, 1138, 8675309]`	`[9035768, 8311, 24, 7988, 321]`
`['hello', 'world', 123, 'orange']`	`['egnaro', 321, 'dlrow', 'olleh']`

10.11.4. Bonus Missions¶

1. Define a function with one parameter, which will be a string. The function must do the following:
    1. Have a clear, descriptive name like `funPhrase`.
    2. Retrieve only the last character from strings with lengths of 3 or less.
    3. Retrieve only the first 3 characters from strings with lengths larger than 3.
    4. Use a template literal to return the phrase `We put the '___' in '___'.` Fill the first blank with the modified string, and fill the second blank with the original string.
    5. Build your function at repl.it.
2. Now test your function:
    1. Outside of the function, define the variable `str` and initialize it with a string (e.g. `'Functions rock!'`).
    2. Call your function and print the returned phrase.
3. The area of a rectangle is equal to its *length x width*.
    1. Define a function with the required parameters to calculate the area of a rectangle.
    2. The function should *return* the area, NOT print it.
    3. Call your area function by passing in two arguments - the length and width.
    4. If only one argument is passed to the function, then the shape is a square. Modify your code to deal with this case.
    5. Use a template literal to print, "The area is ____ cm^2."
    6. Code the area function at repl.it.

Tip

Use these test cases.

1. length = 2, width = 4 (area = 8)
2. length = 14, width = 7 (area = 98)
3. length = 20 (area = 400)

- ← 10.10. Exercises: Functions
- 11. More on Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions

11. More on Functions¶

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.1. Functions as Values

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 characteristics 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 console.log(typeof 42);
2 console.log(typeof "LC101");
3 console.log(typeof Number);
```

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 hello() {
2
3   // function body
4
5 }
6
7 let helloFunc = hello;
```

When a variable refers to a function, we can use the variable name to *call* the function:

```
helloFunc();
```

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 `course` is assigned the value `"LC101"`, then `console.log(course)` 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.

The variable helloFunc on the left *referst to* the function hello on the right

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. More on Functions
- 11.2. Anonymous Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.2. Anonymous Functions

11.2. Anonymous Functions¶

You already know one method for creating a function:

```
function myFunction(parameter1, parameter2,..., parameterN) {

   // function body

}
```

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:

```
function (parameter1, parameter2,..., parameterN) {

   // function body

}
```

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 add = function(a, b) {
2    return a + b;
3 };
4
5 console.log(add(1, 1));
```

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.

The variable add on the left refers to an anonymous function on the right

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 reverse(str) {
2    let lettersArray = str.split('');
3    let reversedLettersArray = lettersArray.reverse();
4    return reversedLettersArray.join('');
5 }
```

repl.it

Question

Consider the code sample below, which declares an anonymous function beginning on line 1.

```
1 let f1 = function(str) {
2    return str + str;
3 };
4
5 let f2 = f1;
```

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 logger = function(errorMsg) {
2    console.log("ERROR: " + errorMsg);
3 };
4
5 if (userInput < 0) {
6    _____________("Invalid input");
7 }
```

repl.it

- ← 11.1. Functions as Values
- 11.3. Passing Functions as Arguments →

Back to top

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 a 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 printMessage() {
2    console.log("The future is now!");
3 }
4
5 setTimeout(printMessage, 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 setTimeout(function () {
2    console.log("The future is now!");
3 }, 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(functionName);
```

When the `map` method executes, the following actions occur:

1. The first element in `someArray` is passed into `functionName` as an argument.
2. `functionName` executes and returns a new value.
3. The return value is added to `mappedArray`.
4. Steps 1 - 3 repeat for each element in `someArray`.

When complete, `mappedArray`, contains each of the individual return values from the mapping function, `functionName`.

Example

```
1 let nums = [3.14, 42, 4811];
2
3 let timesTwo = function (n) {
4    return n*2;
5 };
6
7 let doubled = nums.map(timesTwo);
8
```

```
 9 console.log(nums);
10 console.log(doubled);
```

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 nums = [3.14, 42, 4811];
2
3 let doubled = nums.map(function (n) {
4    return n*2;
5 });
6
7 console.log(doubled);
```

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 nums = [3.14, 42, 4811];
2
3 // TODO: Write a mapping function
4 // and pass it to .map()
5 let halved = nums.map();
6
7 console.log(halved);
```

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.

```
let names = ["Chris", "Jim", "Sally", "Blake", "Paul"];

// TODO: Write a mapping function
// and pass it to .map()
let firstInitials = names.map();

console.log(firstInitials);
```

repl.it

- ← 11.2. Anonymous Functions
- 11.4. Receiving Function Arguments →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.4. Receiving Function Arguments

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 asks the user for some input, using the `prompt` parameter for the text of the question. A second parameter receives a function that does the actual work of validating the input.

Example

```
const input = require('readline-sync');

function getValidInput(prompt, isValid) {

   // Prompt the user, using the prompt string that was passed
```

```
 6    let userInput = input.question(prompt);
 7
 8    // Call the boolean function isValid to check the input
 9    while (!isValid(userInput)) {
10       console.log("Invalid input. Try again.");
11       userInput = input.question(prompt);
12    }
13
14    return userInput;
15 }
16
17 // A boolean function for validating input
18 let isEven = function(n) {
19    return Number(n) % 2 === 0;
20 };
21
22 console.log(getValidInput('Enter an even number:', isEven));
```

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
```

When we call `getValidInput` on line 22, we pass it the string `'Enter an even number:'`, which gets assigned to the `prompt` parameter.

Notice that we also pass in the function `isEven` (with no arguments). This gets assigned to the `isValid` parameter.

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 input = require('readline-sync');
 2
 3 function getValidInput(prompt, isValid) {
 4
 5    let userInput = input.question(prompt);
```

```
 6
 7    while (!isValid(userInput)) {
 8       console.log("Invalid input. Try again.");
 9       userInput = input.question(prompt);
10    }
11
12    return userInput;
13 }
14
15 let isValidPassword = function(password) {
16
17    // Passwords should have at least 8 characters
18    if (password.length < 8) {
19       return false;
20    }
21
22    return true;
23 };
24
25 console.log(getValidInput('Create a password:', isValidPassword));
```

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.

11.4.2.1. A Simple Logger¶

Example

The `logError` function outputs a standardized error message to a location determined by the parameter `logger`.

```
 1 let fileLogger = function(msg) {
 2
 3    // Put the message in a file
 4
 5 }
 6
 7 function logError(msg, logger) {
 8    let errorMsg = 'ERROR: ' + msg;
 9    logger(errorMsg);
10 }
11
12 logError('Something broke!', fileLogger);
```

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.

11.4.2.2. 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 fileLogger = function(msg) {
2
3    // Put the message in a file
4
5 }
```

```
 6
 7 let consoleLogger = function(msg) {
 8
 9    console.log(msg);
10
11 }
12
13 function logError(msg, loggers) {
14
15    let errorMsg = 'ERROR: ' + msg;
16
17    for (let i = 0; i < loggers.length; i++) {
18       loggers[i](errorMsg);
19    }
20
21 }
22
23 logError('Something broke!', [fileLogger, consoleLogger]);
```

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 callMe(func) {
2    func();
3 }
4
5 callMe("Al");
```

repl.it

Question

What type of error occurs when attempting to use a value that is NOT a function as if it were one?

- ← 11.3. Passing Functions as Arguments
- 11.5. Why Use Anonymous Functions? →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.5. Why Use Anonymous Functions?

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 setTimeout(function () {
2    console.log("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.

- ← 11.4. Receiving Function Arguments
- 11.6. Recursion →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.6. Recursion

11.6. Recursion¶

11.6.1. Quick Review¶

In the previous chapter, we learned how to define a function and its parameters.

Example

```
function addTwoToNumber(num){
   return num += 2;
}
```

```
console.log(addTwoToNumber(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.

11.6.1.1. 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 addTwoToNumber(num){
   return num += 2;
}

function addFiveToNumber(value){
   let result = addTwoToNumber(value) + 3;
   return result;
}

console.log(addFiveToNumber(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:

1. Setting up a loop to call one function lots of times, OR
2. 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.

- ← 11.5. Why Use Anonymous Functions?
- 11.7. Recursion Walkthrough: The Base Case →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.7. Recursion Walkthrough: The Base Case

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:

Use a `for` loop to iterate through the array and add each entry into the `newString` variable.

```
let arr = ['L', 'C', '1', '0', '1'];
let newString = '';
```

Use a `while` loop to add the first element in the array to `newString`, then remove that element from the array.

```
let arr = ['L', 'C', '1', '0', '1'];
let newString = '';
```

Use a `for` loop to iterate through the array and add each entry into the `newString` variable.

```
3
4 for (i = 0; i < arr.length; i++){
5    newString = newString + arr[i];
6 }
7
8 console.log(newString);
9 console.log(arr);
```

Console Output

```
'LC101'
['L', 'C', '1', '0', '1']
```

Use a `while` loop to add the first element in the array to `newString`, then remove that element from the array.

```
3
4 while (arr.length > 0){
5    newString += arr[0];
6    arr.shift();
7 }
8 console.log(newString);
9 console.log(arr);
```

Console Output

```
'LC101'
[ ]
```

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 combineEntries(arrayName){
   //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:

```
function combineEntries(arrayName){
   if (baseCase is true){
      //solve last small step
      //end recursion
   } else {
      //call combineEntries again
   }
}
```

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.

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      //call combineEntries again
   }
}
```

`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.

```
function removeI(arr) {
   if (baseCase is true){
      //return final array
      //end recursion
   } else {
      //remove one 'i' entry from array
      //call removeI function again
   }
};
```

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.

```
function factorial(integer) {
   if (baseCase is true){
      //solve last step
      //end recursion
   } else {
      //call factorial function again
   }
};
```

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.

- ← 11.6. Recursion
- 11.8. Making A Function Call Itself →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.8. Making A Function Call Itself

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`:

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      //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']`:

Visual representation for the base case.

Nothing complicated here. `combineEntries` sees only one item in the array, so it returns `'L'`.

Now consider an array with two elements, `['L', 'C']`:

Visual representation for the second-easiest case.

In this case, `combineEntries` executes the `else` statement. We have no code for this yet, but we can still consider the logic:

1. `combineEntries` returns `'L'` and calls itself again using what is left inside the array (`['C']`).
2. When passed `['C']`, which is the base case, `combineEntries` returns `'C'`.
3. The strings `'L'` and `'C'` get combined and returned as the final result.

Next, consider an array with three elements `['L', 'C', '1']`:

Visual representation for the third-easiest case.

As before, `combineEntries` executes the `else` statement, and we can follow the logic:

1. `combineEntries` returns `'L'` and calls itself again using what is left inside the array (`['C', '1']`).
2. When passed `['C', '1']`, `combineEntries` returns `'C'` and calls itself again using what is left inside the array (`['1']`).
3. When passed `['1']`, which is the base case, `combineEntries` returns `'1'`.
4. The strings `'C'` and `'1'` get combined and returned.
5. 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
`'1'`	Base case. Returns the element from an array of length 1.
`'01'`	Combines the first element from an array of length 2 with the base case value.
`'101'`	Combines the first element from an array of length 3 with the two previous values.
`'C101'`	Combines the first element from an array of length 4 with the three previous values.
`'LC101'`	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 slighly 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:

1. Select the first element in the array,
2. 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 combineEntries(arrayName){
2    if (arrayName.length <= 1){
3       return arrayName[0];
4    } else {
5       return arrayName[0]+combineEntries(arrayName.slice(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 `combineEntries(['C', '1', '0', '1'])`.
2	Second call: Combine `'C'`, with `combineEntries(['1', '0', '1'])`.
3	Third call: Combine `'1'`, with `combineEntries(['0', '1'])`.
4	Fourth call: Combine `'0'`, with `combineEntries(['1'])`.
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 decreasingSum(integer) {
2    if (integer === 1){
3       return integer;
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.

- ← 11.7. Recursion Walkthrough: The Base Case
- 11.9. Recursion Wrap-Up →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.9. Recursion Wrap-Up

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:

1. Fewer lines of code required to accomplish a task,
2. Makes code cleaner and more readable.

Drawbacks of Recursion:

1. More abstract than using loops,
2. Code is "more readable" only if the reader understands recursion.

11.9.1. Recursion in a Nutshell¶

1. Build a single function to break a big problem into a slightly smaller version of the *exact same problem*.
2. The function repeatedly calls itself to reduce the problem into smaller and smaller pieces.
3. Eventually, the function reaches a simplest case (the *base*), which it solves.
4. 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.

- ← 11.8. Making A Function Call Itself
- 11.10. Exercises: More on Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.10. Exercises: More on Functions

11.10. Exercises: More on Functions¶

Arrr! Welcome back, pirates.

11.10.1. Practice Your Skills¶

First, create an anonymous function and practice how to use the map method.

1. Create an anonymous function and set it equal to a variable. Your function should:
    1. If passed a number, return the tripled value.

       Check your solution.

    2. If passed a string, return the string "ARRR!"
    3. If NOT passed a number or string, return the data unchanged.

       Check your solution.

    4. Build your function here, and be sure to test it.
2. Add to your code! Use your 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 your work.

11.10.2. Raid a Shuttle¶

You may still be wondering *Why would I ever use an anonymous function?* For today's mission, of course!

You need to hack into the shuttle code and steal supplies. Since the LaunchCode TAs keep a sharp eye on the shuttle goodies, you can't just add new functions like `siphonFuel` or `lootCargo`. You need to be more sneaky.

Clever.

Invisible.

Anonymous.

The first mate swiped a copy of the code you need to hack:

```
function checkFuel(level) {
   if (level > 100000){
      return 'green';
   } else if (level > 50000){
      return 'yellow';
   } else {
      return 'red';
   }
```

```
}

function holdStatus(arr){
   if (arr.length < 7) {
      return `Spaces available: ${7 - arr.length}`;
   } else if (arr.length > 7){
      return `Over capacity by ${arr.length - 7} items.`
   } else {
      return "Full";
   }
}

let fuelLevel = 200000;
let cargoHold = ['meal kits', 'space suits', 'first-aid kit', 'satellit

console.log("Fuel level: "+ checkFuel(fuelLevel));
console.log("Hold status: "+ holdStatus(cargoHold));
```

Hack the code at repl.it.

1. First, steal some fuel from the shuttle:
    1. Define an anonymous function and set it equal to a variable with a normal, non-suspicious name. The function takes one parameter. This will be the fuel level on the shuttle.

       Check your solution.

    2. You must siphon off fuel without alerting the TAs. Inside your function, you want to reduce the fuel level as much as possible WITHOUT changing the color returned by the `checkFuel` function.
    3. Once you figure out how much fuel to pump out, return that value.

       Check your solution.

    4. Decide where to best place your function call to gather our new fuel.

    Tip

    Be sure to test your 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 parameter, and set it equal to another innocent variable.
    2. You 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. You need to replace what you steal with something worthless. The count MUST stay the same, or you'll get caught and thrown into the LaunchCode brig.
4. Don't get hasty, matey! Remember to test your function.

3. Finally, you need to print a receipt for the accountant. Don't laugh! That genius knows MATH and saves us more gold than you can imagine.

    1. Define a function called `irs` that can take `fuelLevel` and `cargoHold` as arguments.

       Check your solution.

    2. Call your anonymous fuel and cargo functions from within `irs`.

    3. Use a template literal to return,
       `"Raided _____ kg of fuel from the tanks, and stole ____ and ____ from the cargo hold."`

       Check your solution.

- ← 11.9. Recursion Wrap-Up
- 11.11. Studio: More Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 11. More on Functions
3. 11.11. Studio: More Functions

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.

11.11.1.1. 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.

```
[ 5, 10, 2, 42 ]
[ -2, 0, -10, -44, 5, 3, 0, 3 ]
[ 200, 5, 4, 10, 8, 5, -3.3, 4.4, 0 ]
```

Code studio part A at repl.it.

11.11.1.2. Part B: Create a New Sorted Array¶

Create another function with an array of numbers as its parameter. Within this function:

1. Define a new, empty array to hold the final sorted numbers.
2. Use your function from the previous exercise to find the minimum value in the old array.
3. Add the minimum value to the new array, and remove the minimum value from the old array.
4. Repeat parts b & c until the old array is empty.
5. Return the new sorted array.
6. *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:

1. Sort each sample array in increasing order.
2. Sort each sample array in decreasing order.
3. Does the function alter `arrayName`?
4. 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 rite 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.

- ← 11.10. Exercises: More on Functions
- 12. Objects and the Math Object →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object

12. Objects and the Math Object¶

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.1. Objects and Why They Matter

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.

Diagram showing that objects are a collection of key/value pairs.

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:

```
let objectName = {key1:value1, key2:value2, key3:value3, ... };
```

If we have a lot of key/value pairs in our object, we can also put each one on a separate line!

```
let objectName = {
    key1: value1,
    key2: value2,
    key3: value3,
    .
    .
    .
};
```

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:

```
let tortoiseOne = {
    species: "Galapagos Tortoise",
    name: "Pete",
    weight: 919,
    age: 85,
    diet: ["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 tortoiseOne = {
 2     species: "Galapagos Tortoise",
 3     name: "Pete",
 4     weight: 919,
 5     age: 85,
 6     diet: ["pumpkins", "lettuce", "cabbage"],
 7     sign: function() {
 8         return this.name + " is a " + this.species;
 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?

1. Objects can store many values
2. Objects have properties
3. Objects have methods
4. Keys are stored as numbers

Question

Which keyword can be used to refer to an object within an object?

1. `Object`
2. `let`
3. `this`

- ← 12. Objects and the Math Object
- 12.2. Working with Objects →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.2. Working with Objects

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.

12.2.1.1. Bracket Syntax¶

To access a property with bracket syntax, the code looks like: `object["key"]`.

12.2.1.2. 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

```
let tortoiseOne = {
    species: "Galapagos Tortoise",
    name: "Pete",
    weight: 919,
    age: 85,
    diet: ["pumpkins", "lettuce", "cabbage"]
};

console.log(tortoiseOne["name"]);
console.log(tortoiseOne.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.

```
let tortoiseOne = {
    species: "Galapagos Tortoise",
    name: "Pete",
    weight: 919,
    age: 85,
    diet: ["pumpkins", "lettuce", "cabbage"]
};

console.log(tortoiseOne.weight);

newWeight = tortoiseOne.weight + 10;

tortoiseOne["weight"] = newWeight;

console.log(tortoiseOne["weight"]);
```

Console Output

```
919
929
```

12.2.2.1. Add New Key/Value Pairs¶

After declaring and initializing an object, we can add new properties at any time by using bracket syntax:

```
objectName["new-key"] = propertyValue;
```

Example

```
let tortoiseTwo = {
    species: "Galapagos Tortoise",
    name: "Pete",
    weight: 919
};

console.log(tortoiseTwo);

tortoiseTwo["age"] = 120;
tortoiseTwo["speed"] = 'Faster than the hare.'

console.log(tortoiseTwo);
console.log(tortoiseTwo.age);
```

Console Output

```
{ species: 'Galapagos Tortoise', name: 'Pete', weight: 919 }
{ species: 'Galapagos Tortoise',
```

```
   name: 'Pete',
   weight: 919,
   age: 120,
   speed: 'Faster than the hare.' }
120
```

12.2.3. Check Your Understanding¶

All of the questions below refer to an object called `giraffe`.

```
let giraffe = {
  species: "Reticulated Giraffe",
  name: "Cynthia",
  weight: 1500,
  age: 15,
  diet: "leaves"
};
```

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;}
```

1. `return`
2. `this`
3. `diet`
4. a comma

Question

Could we use bracket syntax, dot notation, or both to access the properties of `giraffe`?

- ← 12.1. Objects and Why They Matter
- 12.3. Coding With Objects →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.3. Coding With Objects

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!

```
let tortoiseTwo = {
   species: "Galapagos Tortoise",
   name: "Patricia",
   weight: 800,
   age: 85,
   diet: ["pumpkins", "lettuce", "cabbage"],
   sign: function() {
      return this.name + " is a " + this.species;
   }
};
```

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.

```
let tortoiseOne = {
   species: "Galapagos Tortoise",
   diet: ["pumpkins", "lettuce", "cabbage"]
};

let tortoiseTwo = {
   species: "Galapagos Tortoise",
   diet: ["pumpkins", "lettuce", "cabbage"]
};

console.log(tortoiseOne === tortoiseTwo);
```

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

```
let giraffe = {
  species: "Reticulated Giraffe",
  name: "Cynthia",
  weight: 1500,
  age: 15,
  diet: "leaves"
};

for (item in giraffe) {
   console.log(item + ", " + giraffe[item]);
}
```

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

```
let giraffe = {
  species: "Reticulated Giraffe",
  name: "Cynthia",
  weight: 1500,
  age: 15,
  diet: "leaves"
};

function birthday(animal) {
    animal.age = animal.age + 1;
    return animal;
}

console.log(giraffe.age);

birthday(giraffe);

console.log(giraffe.age);
```

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`?

```
let tortoiseOne = {
   age: 150,
   species: "Galapagos Tortoise",
```

```
 4    diet: ["pumpkins", "lettuce", "cabbage"]
 5 };
 6
 7 let tortoiseTwo = {
 8    age: 150,
 9    species: "Galapagos Tortoise",
10    diet: ["pumpkins", "lettuce", "cabbage"]
11 };
```

1. `tortoiseOne == tortoiseTwo`
2. `tortoiseOne === tortoiseTwo`
3. `tortoiseOne.age === tortoiseTwo.age`

- ← 12.2. Working with Objects
- 12.4. The Math Object →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.4. The Math Object

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 console.log(Math.PI);
2 console.log(Math.PI*4);
3 console.log(Math.PI + Math.PI);
```

Console Output

```
3.141592653589793
12.566370614359172
6.283185307179586
```

As stated above, the properties within `Math` *cannot* be changed by the user.

Example

```
1 console.log(Math.PI);
2
3 Math.PI = 1234;
4
5 console.log(Math.PI);
```

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.

- ← 12.3. Coding With Objects
- 12.5. Math Methods →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.5. Math Methods

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.

Ten Common Math Methods¶

Method	Syntax	Description
abs	`Math.abs(number)`	Returns the positive value of `number`.
ceil	`Math.ceil(number)`	Rounds the decimal `number` UP to the closest integer value.
floor	`Math.floor(number)`	Rounds the decimal `number` DOWN to the closest integer value.
max	`Math.max(x,y,z,...)`	Returns the largest value from a set of numbers.
min	`Math.min(x,y,z,...)`	Returns the smallest value from a set of numbers.
pow	`Math.pow(x,y)`	Returns the value of x raised to the power of y (x^y).
random	`Math.random()`	Returns a random decimal value between 0 and 1, NOT including 1.
round	`Math.round(number)`	Returns `number` rounded to the nearest integer value.
sqrt	`Math.sqrt(number)`	Returns the square root of `number`.
trunc	`Math.trunc(number)`	Removes any decimals and returns the integer part of `number`.

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`?

1. `Math.floor(-3.87)`
2. `Math.random(-3.87)`
3. `Math.round(-3.87)`
4. `Math.trunc(-3.87)`

Question

What is printed by the following program?

```
let num = Math.floor(Math.random()*10);

console.log(num);
```

1. A random number between 0 and 9.
2. A random number between 0 and 10.
3. A random number between 1 and 9.
4. A random number between 1 and 10.

Question

What is printed by the following program?

```
let num = Math.round(Math.random()*10);

console.log(num);
```

1. A random number between 0 and 9.
2. A random number between 0 and 10.
3. A random number between 1 and 9.
4. A random number between 1 and 10.

- ← 12.4. The `Math` Object
- 12.6. Combining `Math` Methods →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.6. Combining `Math` Methods

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 randomSelection(arr){
 2    let index = Math.floor(Math.random()*arr.length);
 3    return arr[index];
 4 }
 5
 6 let happiness = ['Hope','Joy','Peace','Love','Kindness','Puppies','Kitt
 7
 8 for (i=0; i < 8; i++){
 9    console.log(randomSelection(happiness));
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*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**
`Math.round(5.56789123*100)/100`	Evaluate the numbers in () first: 5.56789123*100 = 556.789123
`Math.round(556.789123)/100`	Apply the `round` method to 556.789123
`557/100`	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.

Rounding to Decimal Places¶

Decimal Places In Answer	**Multiply & Divide By**	**Syntax**
1	10	`Math.round(number*10)/10`
2	100	`Math.round(number*100)/100`
3	1000	`Math.round(number*1000)/1000`
etc.	etc.	etc.

12.6.3. Check Your Understanding¶

Question

Which of the following correctly rounds 12.3456789 to 4 decimal places?

1. `Math.round(12.3456789)*100/100`
2. `Math.round(12.3456789*100)/100`
3. `Math.round(12.3456789*10000)/10000`
4. `Math.round(12.3456789)*10000/10000`

- ← 12.5. Math Methods
- 12.7. Exercises: Objects & Math →

Back to top

LaunchCode logo

Search

1. Contents
2. 12. Objects and the Math Object
3. 12.7. Exercises: Objects & Math

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

12.7.1. Part 1: Create More Objects¶

Based on the two object literals provided in the starter code, create new object literals for three more animals:

Animal Astronauts¶

Name	Species	Mass (kg)	Age (years)
Brad	Chimpanzee	11	6
Leroy	Beagle	14	5
Almina	Tardigrade	0.0000000001	1

12.7.1.1. Add a New Property¶

For each animal, add a property called `astronautID`. Each `astronautID` should be assigned a number between 1 and 10 (including 10). However, no crew members should have the same ID.

12.7.1.2. Add a Method¶

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. The number can be 0 as well as 10.

12.7.1.3. Store the Objects¶

Create a `crew` array to store all of the animal objects.

Check your solution.

12.7.2. Part 2: Crew Reports¶

Upper management at the space base wants us to report all of the relevant information about the animal astronauts.

Define a `crewReports` function to accomplish this. When passed one of the animal objects, the function returns a template literal with the following string: `'____ is a ____. They are ____ years old and ____ kilograms. Their ID is ____.'`

Fill in the blanks with the name, species, age, mass, and ID for the selected animal.

12.7.3. Part 3: Crew Fitness¶

Before these animal astronauts can get ready for launch, they need to take a physical fitness test. Define a `fitnessTest` function that takes an array as a parameter.

Within the function, race the five animals together by using the `move` method. An animal is done with the race when they reach 20 steps or greater. Store the result as a string: `'____ took ____ turns to take 20 steps.'` Fill in the blanks with the animal's name and race result. Create a new array to store how many turns it takes each animal to complete the race.

Return the array from the function, then print the results to the console (one animal per line).

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 each animal to reach 20 steps.

Check your solution.

- ← 12.6. Combining `Math` Methods
- 12.8. Studio: Objects & Math →

Back to top

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:

1. Select the crew.
2. Perform critical mission calculations.
3. Determine the fuel required for launch.

12.8.1. Select the Crew¶

To access the code for exercise 1, open this repl.it link.

12.8.1.1. Randomly Select ID Numbers¶

Each candidate was assigned an ID number, which is stored in the candidate's data file and in the `idNumbers` array.

1. Write a `selectRandomEntry` 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.
2. Call the function three times 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!

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.

12.8.1.2. Build a `crew` Array¶

Design a function that takes two arrays as parameters. These hold the randomly selected ID numbers and the candidate objects.

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, and then return that array.

Use a template literal to print, `'____, ____, and ____ are going to space!'` Fill in the blanks with the names of the selected animals.

12.8.2. Orbit Calculations¶

To access the code for the orbit calculations and first bonus mission, go to repl.it.

1. 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.

    1. Define a function that returns the circumference ($C = 2\pi r$) of the orbit. Round the circumference to an integer.

    2. Define the `missionDuration` function to take three parameters - the number of orbits completed, the orbit radius, and the orbital speed. Set the default radius to 2000 km and the default orbital speed to 28000 km/hr.

    3. Calculate how long it will take our animals to complete a certain number of orbits (`time = distance/speed`). Round the answer to 2 decimal places, then return the result.

       For example, with the default radius and speed, 5 orbits will take about `2.24 hours`.

    4. Print,
       `'The mission will travel ____ km around the planet, and it will take ____ hours to complete.'`

2. Time for an excursion! Code an `oxygenExpended` function to accomplish the following:

    1. The function should take a candidate object as a parameter and NOT the `crew` array.

       Note

       When you call `oxygenExpended`, feel free to use your `selectRandomEntry` to pick the crew member to pass into the function.

    2. The spacewalk will last for three orbits around the earth. Use `missionDuration` to calculate how many hours the spacewalk will take.

    3. Use the candidate's `o2Used` method to calculate how much oxygen (O_2) they consume during the spacewalk. Round the answer to 3 decimal places.

    4. Return the string, `'__ 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.

5. We should not restrict our mission to the default values for orbital radius and orbital speed. Refactor `oxygenExpended` to accept values for these items. Remember to include the values in the `missionDuration` call.

12.8.3. Bonus Missions¶

12.8.3.1. Conserve O_2¶

Instead of randomly selecting a crew member for the spacewalk, have your program select the animal with the smallest oxygen consumption.

12.8.3.2. Fuel Required for Launch¶

To access the code for this bonus mission, go to repl.it.

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.

1. Write a `crewMass` function that returns the total mass of the selected crew members rounded to 1 decimal place.
2. The mass of the un-crewed rocket plus the food and other supplies is 75,000 kg. Create a `fuelRequired` function to combine the rocket and crew masses, then calculate and return the amount of fuel needed to reach LEO.
3. 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. Update `fuelRequired` to account for this, then round the final amount of fuel UP to the nearest integer.
4. Print `'The mission has a launch mass of ____ kg and requires ____ kg of fuel.'` Fill in the blanks with the calculated amounts.

- ← 12.7. Exercises: Objects & Math
- 13. Modules →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules

13. Modules¶

- ← 12.8. Studio: Objects & Math
- 13.1. What are Modules? →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules
3. 13.1. What are 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:

Visual of the parts of a project.

Imagine if the project included two or three additional programming languages!¶

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.

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. Modules
- 13.2. Require Modules →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules
3. 13.2. Require Modules

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 input = require('readline-sync');
2
```

```
3 let name = input.question("What is your name?");
4 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 input = require('readline-sync');

console.log(typeof input);
```

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.

13.2.2.1. 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:

../../_images/path-example.png

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`.

13.2.2.2. 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 not 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 {
2    "main": "index.js",
3    "dependencies": {
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:

../../_images/requireCC.png

Which statement allows you to import the `rutabaga` module into `project.js`?

1. `const rutabaga = require('/rutabaga.js');`
2. `const rutabaga = require('./rutabaga.js');`
3. `const rutabaga = require('../rutabaga.js');`
4. `const rutabaga = require('./Tubers/rutabaga.js');`

- ← 13.1. What are Modules?
- 13.3. NPM →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules
3. 13.3. NPM

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.

../../_images/readline-sync-npm-results.png

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

../../_images/readline-sync-npm-page.png

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. There is no need to panic! You will learn how to use the terminal in a later chapter.

For now, recall that we coded many Node projects inside of repl.it, which allows us to simulate a development environment WITHOUT having to install any software on our computers. As such, repl.it automatically handles much of the work of installing external modules.

13.3.3. NPM CLI With repl.it¶

Login to your repl.it account, fork this example, and then follow these instructions:

Example

We will use the repl.it interface to add new modules to a project.

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.

   ../../_images/replit-search-for-module.png
4. Verify this is the module you want, then click on the plus icon.

   ../../_images/replit-add-module.png
5. Clicking the plus icon adds a `package.json` file that includes a dependency listing for `readline-sync`.

   ../../_images/replit-package-json-added.png

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 input = require("readline-sync");
2
3 const name = input.question("What is your name?");
4 console.log(`hello ${name}`);
```

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.

- ← 13.2. Require Modules
- 13.4. Exporting Modules →

Back to top

LaunchCode logo

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 isPalindrome(str){
 2    return str === str.split('').reverse().join('');
 3 }
 4
 5 function evenOrOdd(num){
 6    if (num%2===0){
 7       return "Even";
 8    } else {
 9       return "Odd";
10    }
11 }
12
13 function randomArrayElement(arr){
14    let index = Math.floor(Math.random()*arr.length);
15    return arr[index];
16 }
```

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 palindromeCheck = require('./practiceExports.js');
2
3 console.log(typeof palindromeCheck);
4 console.log(palindromeCheck('that'));
5 console.log(palindromeCheck('radar'));
```

Console Output

```
function
false
true
```

There are several points to make about the code and output.

1. Assigning `isPalindrome` to `module.exports` allows us to use that function in other files.
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:

```
module.exports = {
   isPalindrome: isPalindrome,
   evenOrOdd: evenOrOdd,
   randomArrayElement: randomArrayElement
}
```

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 = isPalindrome;
module.exports = evenOrOdd;
module.exports = randomArrayElement;
```

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.

13.4.3.1. 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 practice = require('./practiceExports.js');
2
3 console.log(typeof practice);
4 console.log(practice);
```

`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 practice = require('./practiceExports.js');
2 let arr = ['Hello', 'World', 123, 987, 'LC101'];
3
4 console.log(practice.isPalindrome('mom'));
5 console.log(practice.evenOrOdd(9));
```

```
6
7 for (i=0; i < 3; i++){
8    console.log(practice.randomArrayElement(arr));
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:

1. A file containing JavaScript code intended for use in other Node programs.
2. A separate block of code within a program.
3. One line of code in a program.
4. A function.
5. A file that contains documentation about functions in JavaScript.

Question

Assume you have the following at the end of a `circleStuff.js` module:

```
module.exports = {
   areaOfCircle: areaOfCircle,
   circumference: circumference,
   findRadius: findRadius,
   arcLength: arcLength
}
```

Inside your project, you import `circleStuff`:

```
1 const circleStuff = require('./circleStuff.js');
```

Which of the following is the correct way to find the circumference of a circle from within your project?

1. `circleStuff(argument)`
2. `circleStuff.circumference(argument)`
3. `circleStuff(circumference(argument))`
4. `circumference(argument)`

- ← 13.3. NPM
- 13.5. Wrap-up →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules
3. 13.5. Wrap-up

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.

- ← 13.4. Exporting Modules

- 13.6. Exercises: Modules →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules
3. 13.6. Exercises: Modules

13.6. Exercises: Modules¶

Practice makes better. You will create a project that accomplishes the following:

1. Steps through a list of Yes/No questions.
2. 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.

Check your solution

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.

Check your solution

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.

Check your solution

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.

Check your solution

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.
```


- ← 13.5. Wrap-up
- 13.7. Studio: Boosting Confidence →

Back to top

LaunchCode logo

Search

1. Contents
2. 13. Modules
3. 13.7. Studio: Boosting Confidence

13.7. Studio: Boosting Confidence¶

At this point in your learning journey, it's normal to have doubts about your progress. This class can be challenging. In this studio, we're spending time as a group to discuss our experiences, as well as build our confidence in our coding abilities.

Doubting progress is normal and common. 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.

On your own, spend two minutes gathering your own thoughts and writing down a skill area not related to coding that you have some experience in? Share this with the studio group.

Some examples from your home life:

- Are you an avid home cook?
- Do you know more about how your car runs than your family?
- Are you so good at needlepoint that your friends are asking you to start an Etsy site?

Or examples from your work life:

- Are your presentations so beautiful that your CEO weeps?
- Is your calendar so organized that your coworkers blush when they see it?
- Can you turn any grumpy customer's frown upside down?

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 with your group and your TA(s):

1. Since joining the course, have you ever felt unsure about your future in the tech workforce?
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. Have you ever compared yourself to the students around you? Do you think that this was an objective comparison?
4. What are you most proud of from your time working with this course?
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. Helpful Tips¶

Here are some tips that we recommend 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 this course for a short period of time. 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 programming 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 will not earn you your tech job. There will be LOTS of candidates, and you will shine enough to set you apart. When someone compliments your effort or the quality of your work, graciously accept!
6. *Voice your worth*. Many people use daily exercises to affirm their abilities. Here's an example.
7. *Teach*. This is a great way to reinforce your learning, and it helps you recognize how much you know.
8. *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.

13.7.4. Other Resources¶

- 5 Steps To Shake The Feeling That You're An Impostor
- Stop Telling Women They Have Imposter Syndrome

- ← 13.6. Exercises: Modules
- 14. Unit Testing →

Back to top

LaunchCode logo

Search

1. Contents
2. 14. Unit Testing

14. Unit Testing¶

- ← 13.7. Studio: Boosting Confidence
- 14.1. Why Test Your Code? →

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 a 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 is called a **unit-testing framework**, 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 and used in a manner similar to `readline-sync`. Usually, Jasmine must be manually installed into a project, but we do not need to learn how to do this yet.

In this chapter we will continue to use repl.it, which automatically installs npm modules when it runs a program that contains a `require` statement.

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:

A small project with three important files: index.js, reverse.js, reverse.spec.js

A Jasmine project¶

There are three important files:

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 (like 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.

14.2.2.1. `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 = require('jasmine');
 2 const jasmine = new Jasmine();
 3
 4 jasmine.loadConfig({
 5    spec_dir: 'spec',
 6    spec_files: [
 7       "**/*[sS]pec.js"
 8    ],
 9 });
10
11 jasmine.execute();
```

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.

14.2.2.2. `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.

Repl.it menu button for creating new files

Name the new file `hello.js`, then add this code:

```
1 function hello(name) {
2    if (name === undefined)
3       name = "World";
4
5    return "Hello, " + name + "!";
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.

```
module.exports = hello;
```

14.2.2.3. `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:

File tree for test project

At the top of the `hello.spec.js` file, import your function from `hello.js`:

```
1 const hello = require('../hello.js');
```

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.

```
describe("hello", function(){

});
```

14.2.2.4. Specifications and Expectations¶

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:

```
it("should return custom message when name is specified", function(){
   expect(hello("Jasmine")).toEqual("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 desired behavior. This string serves to document the test and is also used in reporting test results. These strings will usually begin with "should", followed by a desired action.

The second argument to `it` is yet another anonymous function. This function contains the test code itself, which takes the form of an **expectation**. An expectation is a declaration of desired behavior *in code*. Let's examine the contents of the anonymous function:

```
expect(hello("Jasmine")).toEqual("Hello, Jasmine!");
```

Calling `expect(x).toEqual(y)` declares that we expect `x` to equal `y`. As you get started with unit testing, nearly *all* of your tests will take this form. The argument to `expect()` is a call to the function `hello()`. The argument to `toEqual()` is the expected output from that function call. `toEqual()` is a specialized method called a **matcher**. Matchers in Jasmine compare the value passed to the value passed to `expect()`. These comparisons are not just limited to checking if the two values are equal. Jasmine has a wide variety of matchers built-in and developers can also build custom matchers. For a full list of the provided matchers, check out the Jasmine documentation.

If the two arguments are indeed equal, the test will pass. Otherwise, the test will fail. In this case, we are declaring that we *expect* `hello("Jasmine")` to return the value `"Hello, Jasmine!"`.

Your whole test file should now look like this:

```
1 const hello = require('../hello.js');
2
3 describe("hello world test", function(){
4
5    it("should return a custom message when name is specified", function(
6       expect(hello("Jasmine")).toEqual("Hello, Jasmine!");
7    });
8
9 });
```

14.2.2.5. 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
6 1 spec, 0 failures
7 Finished in 0.016 seconds
8 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. If our test had failed, then the line would have read:

```
1 spec, 1 failure
```

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, " + name;
```

Run the tests again. This time, the output looks quite different:

```
Randomized with seed 41448
Started
```

```
F

Failures:
1) hello world test should return a custom message when name is specified
   Message:
      Expected 'Hello, Jasmine' to equal 'Hello, Jasmine!'.
   Stack:
      Error: Expected 'Hello, Jasmine' to equal 'Hello, Jasmine!'.
         at <Jasmine>
         at UserContext.<anonymous> (/home/runner/Hello-Jasmine-Expectatio
         at <Jasmine>

1 spec, 1 failure
Finished in 0.01 seconds
Randomized with seed 41448 (jasmine --random=true --seed=41448)
```

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.

```
it("should return a general greeting when name is not specified", function
     expect(hello()).toEqual("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.

14.2.3. Check Your Understanding¶

Question

Examine the function below, which checks if two strings match:

```
function doStringsMatch(string1, string2){
   if (string1 === string2) {
      return 'Strings match!';
   } else {
      return 'No match!';
   }
}
```

Which of the following tests checks if the function properly handles case-*sensitive* answers.

1. `expect(doStringsMatch('Flower', 'Flower')).toEqual('Strings match!');`
2. `expect(doStringsMatch('Flower', 'flower')).toEqual('No match!');`
3. `expect(doStringsMatch('Flower', 'plant')).toEqual('No match!');`
4. `expect(doStringsMatch('Flower', '')).toEqual('No match!');`

- ← 14.1. Why Test Your Code?
- 14.3. Unit Testing in Action →

Back to top

LaunchCode logo

Search

1. Contents
2. 14. Unit Testing
3. 14.3. Unit Testing in Action

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 reverse(str) {
2    return str.split('').reverse().join('');
3 }
4
5 function isPalindrome(str) {
6    return reverse(str) === str;
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:

```
module.exports = isPalindrome;
```

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 isPalindrome = require('../palindrome.js');
2
3 describe("isPalindrome", function(){
4
5   // TODO - write some tests!
6
7 });
```

Okay, let's write some tests!

14.3.3. Positive and Negative Test Cases¶

14.3.3.1. 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 isPalindrome = require('../palindrome.js');
 2
 3 describe("isPalindrome", function(){
 4
 5    it("should return true for a single letter", function(){
 6       expect(isPalindrome("a")).toBeTrue();
 7    });
 8
 9    it("should return true for a single letter repeated", function(){
10       expect(isPalindrome("aaa")).toBeTrue();
11    });
12
13    it("should return true for a simple palindrome", function(){
14       expect(isPalindrome("aba")).toBeTrue();
15    });
16
17    it("should return true for a longer palindrome", function(){
18       expect(isPalindrome("racecar")).toBeTrue();
19    });
20
21 });
```

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.

14.3.3.2. 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.

```
 1 it("should return false for a longer non-palindrome", function(){
 2    expect(isPalindrome("launchcode")).toBeFalse();
 3 });
 4
 5 it("should return false for a simple non-palindrome", function(){
 6    expect(isPalindrome("ab")).toBeFalse();
 7 });
 8
 9 it("should be case-sensitive", function(){
10    expect(isPalindrome("abA")).toBeFalse();
11 });
12
13 it("should consider whitespace", function(){
14    expect(isPalindrome("so many dynamos")).toBeFalse();
15 });
```

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:

```
it("should consider the empty string a palindrome", function(){
   expect(isPalindrome("")).toBeTrue();
});
```

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?

1. `aa`
2. `aBa`
3. `Mom`
4. `Taco Cat`
5. `AbAb`

Question

Which of the *negative* test cases listed above are no longer valid for our case-insensitive `isPalindrome`?

1. `ab`
2. `launchcode`
3. `abA`
4. `so many dynamos`

- ← 14.2. Hello, Jasmine!
- 14.4. Test-Driven Development →

Back to top

LaunchCode logo

Search

1. Contents
2. 14. Unit Testing
3. 14.4. Test-Driven Development

14.4. Test-Driven Development¶

Now that we know more about unit tests, 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. Each test must *clearly describe* the behavior it is testing.

Example

Example test case for a data parsing project:

- Take in a 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.

- The test case will evaluate the function named `parseData`
- The `parseData` function will:
  - contain a `data` parameter that gets assigned a string of data
  - contain a `delimiter` parameter that will be used to split the string into an 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 as `expect().toEqual()` to clearly demonstrate that the proposed new code functions properly.

Example

Next, type out the ideas into an actual 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 `expect().toBeTrue()` that verifies an array is returned.

```
const parse = require('../parse-numbers');

describe("parse numbers", function(){

   it("returns array when passed comma separated list of numbers", func
      let items = parse("5,8,0,17,6,4,9,3", ",");
      expect(Array.isArray(items)).toBeTrue();
   });

});
```

Now run the test! The test should fail (or not compile at all) 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 new test pass, a file must be created that exports a `parseData` function with logic that satisfies the expected result.

```
function parseData(text, delimiter) {
   return text.split(delimiter);
}

module.exports = parseData;
```

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.

   Graphic showing the cycle of phases from red the writing test, green making the test pass, and blue of refactoring code to be better which points back to red.

   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 error will be immediately identified by the test. Here are a few examples of refactoring:

1. Using different data structures,
2. Reducing the number of times needed to loop through an array,
3. Moving duplicate logic into a function so it can be reused.

The refactor is also done in a TDD process:

1. Decide how to improve the implementation of the feature,
2. Change the unit test to use this new idea,
3. Run the code to see the test fail,
4. Refactor the code to implement the new idea,
5. Finally, see the test pass with the refactored design.

- ← 14.3. Unit Testing in Action
- 14.5. TDD in Action →

Back to top

LaunchCode logo

Search

1. Contents
2. 14. Unit Testing
3. 14.5. TDD in Action

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

```
{
   id: 1410,
   rawData: 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:

1. Create a blank test function.
2. 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 describe("transmission processor", function() {
2
3   it("", function(){
4
5   });
6
7 });
```

Give the test the name `"takes a string and returns an object"`.

```
1 describe("transmission processor", function() {
2
3   it("takes a string returns an object", function() {
4
5   });
6
7 });
```

Now that we identified a clear goal for the test, let's add logic and `expect` calls in the test to verify the desired behavior. *But wait...* we haven't added anything except an empty test 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 and 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 processor = require('../processor.js');
2
3 describe("transmission processor", function() {
4
5   it("takes a string and returns an object", function(){
6
7   });
8
9 });
```

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 `expect().toEqual()` to verify that if we pass a string to `processor`, an object is returned. Carefully review lines 7 and 8 shown below.

```
 1 const processor = require('../processor.js');
 2
 3 describe("transmission processor", function() {
 4
 5    it("takes a string and returns an object", function(){
 6       let result = processor("9701::<489584872710>");
 7       expect(typeof result).toEqual("object");
 8    });
 9
10 });
```

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.

14.5.2.1. 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'
```

14.5.2.2. Go Green!¶

Now that we have a failing test, we have only one choice. Make it pass.

1. Add a `processor.js` file to your repl.it.
2. 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 processor(transmission) {
2    return {};
3 }
4
5 module.exports = processor;
```

Screen shot showing processor.js file added to replit with function in it.

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
```

14.5.2.3. 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 the 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 processor = require('../processor.js');
 2
 3 describe("transmission processor", function() {
 4
 5    it("takes a string and returns an object", function(){
 6       let result = processor("9701::<489584872710>");
 7       expect(typeof result).toEqual("object");
 8    });
 9
10    it("returns -1 if '::' not found", function(){
11       let result = processor("9701<489584872710>");
12       expect(result).toEqual(-1);
13    });
```

```
14
15 });
```

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 and empty object, `{}`.

```
Failures:
1) transmission processor returns -1 if '::' not found
   Message:
      Expected Input A to equal Input B
```

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.indexOf("::") < 0) {
3      // Data is invalid
4      return -1;
5   }
6   return {};
7 }
8
9 module.exports = processor;
```

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 `not.toEqual()` was used, which is checking if the two values are NOT equal. `not.toEqual()` 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.

```
const processor = require('../processor.js');

describe("transmission processor", function() {

   it("takes a string returns an object", function(){
      let result = processor("9701::<489584872710>");
      expect(typeof result).toEqual("object");
   });

   it("returns -1 if '::' not found", function(){
      let result = processor("9701<489584872710>");
      expect(result).toEqual(-1);
   });

   it("returns id in object", function() {
     let result = processor("9701::<489584872710>");
     expect(result.id).not.toEqual(undefined);
   });

});
```

The fail message looks a little different than what we have seen. The phrase "Expected 'actual' to be strictly unequal to" 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:
      Expected "actual" to not equal 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.

```
function processor(transmission) {
   if (transmission.indexOf("::") < 0) {
      // Data is invalid
      return -1;
   }
   let parts = transmission.split("::");
   return {
      id: parts[0]
   };
}

module.exports = processor;
```

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 it("converts id to a number", function() {
2    let result = processor("9701::<489584872710>");
3    expect(result.id).toEqual(9701);
4 });
```

Fail Message

```
Failures:
   1) transmission processor converts id to a number
      Message:
         Expected '9701' to equal 9701.
```

Convert the id part of the string to be of type `number`.

```
 1 function processor(transmission) {
 2    if (transmission.indexOf("::") < 0) {
 3       // Data is invalid
 4       return -1;
 5    }
 6    let parts = transmission.split("::");
 7    return {
 8       id: Number(parts[0])
 9    };
10 }
11
12 module.exports = processor;
```

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 it("returns rawData in object", function() {
2    let result = processor("9701::<487297403495720912>");
3    expect(result.rawData).not.toEqual(undefined);
4 });
```

Fail Message

```
Failures:
1) transmission processor returns rawData in object
   Message:
      Expected "actual" to not equal 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 processor(transmission) {
 2    if (transmission.indexOf("::") < 0) {
 3       // Data is invalid
 4       return -1;
 5    }
 6    let parts = transmission.split("::");
 7    let rawData = parts[1];
 8    return {
 9       id: Number(parts[0]),
10       rawData: rawData
11    };
12 }
13
14 module.exports = processor;
```

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 the symbol is in the wrong place
5. Has > but the symbol is in the wrong place

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 it("returns -1 for rawData if missing < at position 0", function() {
2    let result = processor("9701::487297403495720912>");
3    expect(result.rawData).toEqual(-1);
4 });
```

Fail Message

```
Failures:
1) transmission processor returns -1 for rawData if missing < at position
   Message:
      Expected values to be equal
```

Now add new code to `processor.js` to make the 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 processor(transmission) {
 2    if (transmission.indexOf("::") < 0) {
 3       // Data is invalid
 4       return -1;
 5    }
 6    let parts = transmission.split("::");
 7    let rawData = parts[1];
 8    if (rawData[0] !== "<") {
 9       rawData = -1;
10    }
11    return {
12       id: Number(parts[0]),
13       rawData: rawData
14    };
```

```
15 }
16
17 module.exports = processor;
```

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 features below. 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` cannot be converted to a number.
3. Return `-1` if more than one `"::"` is found in `transmission`.
4. Do not include the `< >` symbols in the value assigned to `rawData`.
5. Return `-1` for the value of `rawData` if anything besides numbers are present between the `< >` symbols.

- ← 14.4. Test-Driven Development
- 14.6. Exercises: Unit Testing →

Back to top

LaunchCode logo

Search

1. Contents
2. 14. Unit Testing
3. 14.6. Exercises: Unit Testing

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 checkFive(num){
2    let result = '';
3    if (num < 5){
4       result = num + " is less than 5.";
5    } else if (num === 5){
6       result = num + " is equal to 5.";
7    } else {
8       result = num + " is greater than 5.";
9    }
10
11   return result;
12 }
```

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:

1. `checkFive.js`, which holds the code for the function,
2. `checkFive.spec.js`, which will hold the testing code,
3. `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.

    1. `checkFive.spec.js` needs to access `checkFive.js`. Add a `require` statement to accomplish this (review Unit Testing in Action if needed).

        Check your solution.

    2. 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 = require('../checkFive.js');

describe("checkFive", function(){

   it("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*.

    1. 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 Expectations section to review best practices.

       Check your solution.

    2. Define the variable `output`, and initialize it by passing a value of 2 to `checkFive`.

```
const checkFive = require('../checkFive.js');

describe("checkFive", function(){

   it("Descriptive feedback...", function(){
      let output = checkFive(2);
   });

});
```

    3. Now use the `expect` function to check the result:

```
const checkFive = require('../checkFive.js');

describe("checkFive", function(){

   it("Descriptive feedback...", function(){
      let output = checkFive(2);
      expect(output).toEqual("2 is less than 5.");
```

```
 8    });
 9
10});
```

Check your solution.

4. 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
```

5. 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 numb
Message:
   Expected Input A to equal Input B:
```

6. 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 whoWon(player1,player2){
2
3    if (player1 === player2){
4       return 'TIE!';
5    }
6
7    if (player1 === 'rock' && player2 === 'paper'){
```

```
 8       return 'Player 2 wins!';
 9    }
10
11    if (player1 === 'paper' && player2 === 'scissors'){
12       return 'Player 2 wins!';
13    }
14
15    if (player1 === 'scissors' && player2 === '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.

   Check your solution.

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.

   Check your solution.

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.

- ← 14.5. TDD in Action
- 14.7. Studio: Unit Testing →

Back to top

LaunchCode logo

Search

1. Contents

14.7. Studio: Unit Testing¶

LaunchCode's Marketing Team needs your help! To make the website more maintainable, they thought an object called `launchcode` that contains important facts and functions they need would be helpful. This way if they need to make a change to one of the facts, they just have to change the object in one place as opposed to going through pages of code to make the change in every place.

Here is what they need the `launchcode` object to contain:

1. A key called `organization` with a value of `"nonprofit"`.
2. A key called `executiveDirector` with a value of `"Jeff"`.
3. A key called `percentageCoolEmployees` with a value of `100`.
4. A key called `programsOffered` with a value of `["LC101", "LaunchCode Women+", "CodeCamp"]`.
5. And a method called `launchOutput()`. This method will return a string.

Let's use Test-Driven Development to write this code! 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 and note the files:

1. `index.js` holds the object we want to design.
2. `launchcode.spec.js` holds the testing script.

The files are mostly empty. Only a framework has been provided for you.

14.7.2. Start With the Properties¶

Let's start our work with the properties we need to add to the object.

14.7.2.1. `organization`¶

1. Inside the `describe` function in `launchcode.spec.js`, write a test that will check that the value of `organization` is `"nonprofit"`. Run your test.

2. With your test complete, turn your attention to `launchcode.js` and add the `organization` property to `launchcode`.
3. Run your tests to make sure that everything works as expected.

14.7.2.2. `executiveDirector`¶

1. Write a new test that will check that the value of `executiveDirector` is `"Jeff"`. Run your test.
2. Add the `executiveDirector` property to `launchcode`.
3. Run your tests!

14.7.2.3. `percentageCoolEmployees`¶

1. Write another test that will check that the value of `percentageCoolEmployees`. Run your test.
2. Add `percentageCoolEmployees` to `launchcode`.
3. Run your tests!

14.7.2.4. `programsOffered`¶

1. Write a fourth test that will check the value of `programsOffered`. You should have four `expect` statements within your test. Three of them should check that the array contains the appropriate values and the final one should check that the array is the appropriate size. Before moving on, take a moment either individually or with a classmate, to reflect on why you would need these four `expect` statements. Run your test.
2. Add the `programsOffered` property to `launchcode`.
3. Run your tests!

You now have the properties set up for the `launchcode` object. Time to move on to creating the `launchOutput()` method.

14.7.3. `launchOutput()`¶

`launchOutput()` needs to meet 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.'`

To make sure that you meet all of these conditions, you need to take it one test at a time.

14.7.3.1. Write the First Test¶

In `launchcode.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!

14.7.3.2. 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.

14.7.3.3. Write the Next Two Tests¶

In `launchCodeRocks.spec.js`, add tests for the conditions:

1. When passed a number that is ONLY divisible by 3, `launchOutput()` returns `'Code!'`
2. When passed a number that is ONLY divisible by 5, `launchOutput()` returns `'Rocks!'`

Run the tests. The two new ones should fail, but the first should still pass. Modify the `it` statements as needed if you see a different result.

14.7.3.4. 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.

14.7.3.5. Hmmm, Tricky¶

In `launchCodeRocks.spec.js`, add a test for the condition:

> 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.

14.7.3.6. 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.

14.7.4. 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.

14.7.5. Bonus Missions¶

14.7.5.1. 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.

- ← 14.6. Exercises: Unit Testing
- 15. Scope →

Back to top

LaunchCode logo

Search

1. Contents
2. 15. Scope

15. Scope¶

Back to top

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

```
let a = 0;

function coolFunction() {
   let b = 2;
   return a + b;
}
```

`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

```
let a = 0;
console.log(a);

function coolFunction() {
```

```
 5    let b = 2;
 6    console.log(`a = ${a}, b = ${b}.`);
 7    return a + b;
 8 }
 9
10 a += 1;
11 console.log(a);
12
13 coolFunction();
14 console.log(b);
```

Console Output

```
0
1
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 myFunction() {
2    let i = 10;
3    return 10 + i;
4 }
5
6 console.log(i);
```

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

```
// Code here CAN use newVariable.

function coolFunction() {
   newVariable = 5;
   return newVariable;
}

// Code here CAN use newVariable.
```

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:

```
let a = 0;

function coolFunction() {
   let b = 2;
   return a + b;
}

function coolerFunction() {
   let c = 5;
   c += coolFunction();
```

```
11    return c;
12 }
13
14 console.log(coolFunction());
15 console.log(coolerFunction());
```

Now, let's consider the execution context for each step.

1. First, the global execution context is entered as the compiler executes the code.

   Figure showing global execution context at the bottom of the stack.

2. Once `coolFunction()` is hit, the compiler creates and executes `coolFunction()` under the `coolFunction()` execution context.

   Figure showing coolFunction on top of global execution context.

3. Upon completion, the compiler returns to the global execution context.

   Figure showing global execution context at the bottom of the stack.

4. The compiler stays at the global execution context until the creation and execution of `coolerFunction()`.

   Figure showing coolerFunction on top of the global execution context.

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.

   Figure showing coolFunction on top of coolerFunction on top of the global execution context.
   Figure showing coolerFunction on top of the global execution context.
   Figure showing global execution context at the bottom of the stack.

15.1.4. Check Your Understanding¶

Both of the concept checks refer to the following code block:

```
1 function myFunction(n) {
2    let a = 100;
3    return a + n;
4 }
5
6 let x = 0;
7
8 x = myFunction(x);
```

Question

What scope is variable x?

1. Global
2. Local

Question

In what order will the compiler execute the code?

- ← 15. Scope
- 15.2. Using Scope →

Back to top

LaunchCode logo

Search

1. Contents
2. 15. Scope
3. 15.2. Using Scope

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 input = require('readline-sync');
2
3 function hello(name) {
4    console.log('Hello,', name);
5    name = 'Ruth';
6    return doubleName(name);
7 }
8
9 function doubleName(name){
```

```
10    console.log(name+name);
11    return name+name;
12 }
13
14 let name = input.question("Please enter your name: ");
15
16 hello(name);
17 doubleName(name);
18 console.log(name);
```

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?

1. `let`
2. `var`
3. `const`

Question

Consider this code:

```
1 let a = 0;
2
```

```
function myFunction() {
   let a = 10;
   return a;
}
```

Because there are two separate variables with the name, `a`, under the two different scopes, `a` is being shadowed.

1. True
2. False

- ← 15.1. Introduction
- 16. More on Types →

Back to top

LaunchCode logo

Search

1. Contents
2. 16. More on Types

16. More on Types¶

- ← 15.2. Using Scope
- 16.1. Primitive Data Types →

Back to top

LaunchCode logo

Search

1. Contents
2. 16. More on Types
3. 16.1. Primitive Data 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 x;
2 console.log(x)
```

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 x = null;
console.log(x);
```

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:

```
let giraffe = {
   species: "Reticulated Giraffe",
   name: "Cynthia",
   weight: 1500,
   age: 15,
   diet: "leaves"
};
```

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:

```
let giraffeTwo = {
   species: "Reticulated Giraffe",
   name: "Alicia",
   weight: null,
   age: 10,
   diet: "leaves"
};
```

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.

1. arrays
2. Strings
3. objects
4. `null`

Question

Consider the following code block:

```
let x;

console.log(x);
```

x is of what data type?

1. `null`

2. `undefined`
3. `NaN`
4. number

- ← 16. More on Types
- 17. Exceptions →

Back to top

LaunchCode logo

Search

1. Contents
2. 17. Exceptions

17. Exceptions¶

- ← 16.1. Primitive Data Types
- 17.1. Introduction →

Back to top

LaunchCode logo

Search

1. Contents
2. 17. 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

```
console.log("This is" an example);
```

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

```
console.log(x[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 a = "Launch";
2
3 a = "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 'a'`.

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 JavaScript, 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. Exceptions
- 17.2. Throw →

Back to top

LaunchCode logo

Search

1. Contents
2. 17. Exceptions
3. 17.2. Throw

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

```
1 throw Error("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 as thrown from line 1, which we know is true because our example only has one line of code.

Note

With all that we have learned about unit testing, you might be wondering how you test if an error is thrown when it should be. To do so, let's imagine our example above is inside a function called `checkThrow()`. We need can then use the `toThrow()` matcher like so:

```
expect( function() {
   checkThrow();
}).toThrow(new Error('You cannot divide by zero!'));
```

17.2.2. Pre-existing Error¶

JavaScript also gives us the power to throw a more specific type of error.

Example

```
throw SyntaxError("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?

- ← 17.1. Introduction

- 17.3. Exceptions as Control Flow →

Back to top

LaunchCode logo

Search

1. Contents
2. 17. Exceptions
3. 17.3. Exceptions as Control Flow

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 input = require('readline-sync');
 2
 3 let animals = [{name: 'cat'}, {name: 'dog'}];
 4 let index = Number(input.question("Enter index of animal:"));
 5
 6 try {
 7    console.log('animal at index:', animals[index].name);
 8 } catch(err) {
 9    console.log("We caught a TypeError, but our program continues to run
10    console.log("You tried to access an animal at index:", index);
11 }
12
13 console.log("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...
```

On line 8 of the above code sample, a variable called `err` is passed to `catch`. `err` can be any type of error object in JavaScript. For the above example, that is a `TypeError`, but we could have had a `ReferenceError` or `EvalError`. If necessary, we could output different statements based on what kind of error `err` is.

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 input = require('readline-sync');
 2
 3 let animals = [{name: 'cat'}, {name: 'dog'}];
 4 let index = Number(input.question("Enter index of animal:"));
 5
 6 try {
 7    console.log('animal at index:', animals[index].name);
 8 } catch(err) {
 9    console.log("We caught a TypeError, but our program continues to run
10 } finally {
11    console.log("You tried to access an animal at index:", index);
12 }
13
14 console.log("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`

- ← 17.2. Throw
- 17.4. Exercises: Exceptions →

Back to top

LaunchCode logo

Search

1. Contents
2. 17. Exceptions
3. 17.4. Exercises: Exceptions

17.4. Exercises: Exceptions¶

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 `denominator` is zero you should throw the error, `"Attempted to divide by zero."`

Note

Hint: You can use an `if / throw` statement to complete this exercise.

Code your function at this repl.it.

Check your solution.

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.

```
function gradeLabs(labs) {
  for (let i=0; i < labs.length; i++) {
     let lab = labs[i];
     let result = lab.runLab(3);
     console.log(`${lab.student} code worked: ${result === 27}`);
  }
}

let studentLabs = [
  {
     student: 'Carly',
     runLab: function (num) {
        return Math.pow(num, num);
     }
  },
  {
     student: 'Erica',
     runLab: function (num) {
        return num * num;
     }
  }
];

gradeLabs(studentLabs);
```

repl.it

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.

```
let studentLabs2 = [
  {
     student: 'Blake',
     myCode: function (num) {
        return Math.pow(num, num);
     }
  },
```

```
 8    {
 9       student: 'Jessica',
10       runLab: function (num) {
11          return Math.pow(num, num);
12       }
13    },
14    {
15       student: 'Mya',
16       runLab: function (num) {
17          return num * num;
18       }
19    }
20 ];
21
22 gradeLabs(studentLabs2);
```

Upon running the second example, the teacher gets `TypeError: lab.runLab is not a function`.

Add a `try/catch` block inside of `gradeLabs` to catch an exception if the `runLab` property is not defined. If the exception is thrown, `result` should be set to the text `"Error thrown"`.

- ← 17.3. Exceptions as Control Flow
- 17.5. Studio: Strategic Debugging →

Back to top

LaunchCode logo

Search

1. Contents
2. 17. Exceptions
3. 17.5. Studio: Strategic Debugging

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.

- ← 17.4. Exercises: Exceptions
- 18. Classes →

Back to top

LaunchCode logo

Search

1. Contents
2. 18. Classes

18. Classes¶

- ← 17.5. Studio: Strategic Debugging
- 18.1. What Are Classes? →

Back to top

LaunchCode logo

Search

1. Contents
2. 18. Classes
3. 18.1. What Are 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:

```
let fox = {
   name: 'Fox',
   age: 7,
   mass: 12,
   catchPhrase: function(repeats) {
      let phrase = 'LaunchCode';
      for (let i = 0; i < repeats; i++) {
         phrase += ' Rocks';
      }
      return phrase;
   }
}

console.log(`${fox.name} is ${fox.age} years old and has a mass of ${fo
console.log(`${fox.name} says, "${fox.catchPhrase(3)}."`);
```

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.

Visual of the relationship between classes and objects.

- ← 18. Classes
- 18.2. Declaring and Calling a Class →

Back to top

LaunchCode logo

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:

```
class ClassName {
   constructor(parameters) {
      //assign properties
   }
   //define methods
}
```

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.

18.2.1.1. 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:

```
class Astronaut {
   constructor(name, age, mass) {
      this.name = name;
      this.age = age;
      this.mass = mass;
   }
}
```

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 objectName = new ClassName(arguments);
```

`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 may differ.

Example

Let's create objects for two of our crew members: Fox and Hippo.

```
 1 class Astronaut {
 2    constructor(name, age, mass){
 3       this.name = name;
 4       this.age = age;
 5       this.mass = mass;
 6    }
 7 }
 8
 9 let fox = new Astronaut('Fox', 7, 12);
10 let hippo = new Astronaut('Hippo', 25, 1500);
11
12 console.log(typeof hippo, typeof fox);
13
14 console.log(hippo, fox);
```

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.

Note

Two objects created from the same class are NOT equal, even if the keys within the objects all have the same values. The reason behind this was discussed previously.

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 Astronaut {
 2    constructor(name, age, mass){
 3       this.name = name;
 4       this.age = age;
 5       this.mass = mass;
 6    }
 7 }
 8
 9 let fox = new Astronaut('Fox', 7, 12);
10
11 console.log(fox);
12 console.log(fox.age, fox.color);
13
14 fox.age = 9;
15 fox.color = 'red';
16
17 console.log(fox);
18 console.log(fox.age, fox.color);
```

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`.

18.2.2.1. Setting Default Values¶

What happens if we create a new `Astronaut` without passing in all of the required arguments?

Try It!

```
 1 class Astronaut {
 2    constructor(name, age, mass){
 3       this.name = name;
 4       this.age = age;
 5       this.mass = mass;
 6    }
 7 }
 8
 9 let tortoise = new Astronaut('Speedy', 120);
10
11 console.log(tortoise.name, tortoise.age, tortoise.mass);
```

repl.it

To avoid issues with missing arguments, we can set a *default* value for a parameter as follows:

```
1 class Astronaut {
2    constructor(name, age, mass = 54){
3       this.name = name;
4       this.age = age;
5       this.mass = mass;
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`.

```
class Car {
   constructor(make, model, year, color, mpg){
      this.make = make;
      this.model = model;
      this.year = year;
      this.color = color;
      this.mpg = mpg;
   }
}
```

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)`?

1. object
2. string
3. function
4. number
5. property

Question

If we call the class with `let myCar = new Car('Tesla', 'Model S', 2019)`, what is output by `console.log(myCar)`?

1. Car {make: 'Tesla', model: 'Model S', year: 2019, color: undefined, mpg: undefined }
2. Car {make: 'Tesla', model: 'Model S', year: 2019, color: '', mpg: '' }
3. Car {make: 'Tesla', model: 'Model S', year: 2019 }

- ← 18.1. What Are Classes?
- 18.3. Assigning Class Methods →

Back to top

LaunchCode logo

Search

1. Contents
2. 18. Classes
3. 18.3. Assigning Class Methods

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.

```
class ClassName {
   constructor(parameters) {
      //assign properties with this.key = value
   }

   methodName(parameters) {
      //function code
   }
}
```

Example

```
class Astronaut {
   constructor(name, age, mass){
      this.name = name;
      this.age = age;
      this.mass = mass;
   }

   reportStats() {
      let stats = `${this.name} is ${this.age} years old and has a mass
      return stats;
   }
}

let fox = new Astronaut('Fox', 7, 12);
console.log(fox.reportStats());
```

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.

```
class ClassName {
   constructor(parameters) {
      this.methodName = function(parameters) {
         //function code
      }
   }
}
```

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:

```
class Astronaut {
   constructor(name, age, mass){
      this.name = name;
      this.age = age;
      this.mass = mass;
      this.reportStats = function() {
         let stats = `${this.name} is ${this.age} years old and has a m
         return stats;
      }
   }
}

let fox = new Astronaut('Fox', 7, 12);

console.log(fox.reportStats());
```

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.

```
// Here we assign the method inside the constructor
class AstronautI {
   constructor(name, age, mass){
      this.name = name;
      this.age = age;
      this.mass = mass;
      this.reportStats = function() {
         let stats = `${this.name} is ${this.age} years old and has a m
         return stats;
      }
   }
}

// Here we assign the method outside of the constructor
class AstronautO {
   constructor(name, age, mass){
      this.name = name;
      this.age = age;
      this.mass = mass;
   }

   reportStats() {
      let stats = `${this.name} is ${this.age} years old and has a mass
      return stats;
   }
}

let fox = new AstronautI('Fox', 7, 12);
let hippo = new AstronautO('Hippo', 25, 1000);

console.log(fox);
console.log(hippo);
```

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?

```
class Plant {
   constructor(type, height) {
      this.type = type;
      this.height = height;
   }

   grow  {
      this.height = this.height + 1;
   }
}
```


- ← 18.2. Declaring and Calling a Class
- 18.4. Inheritance →

Back to top

LaunchCode logo

Search

1. Contents
2. 18. Classes
3. 18.4. Inheritance

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.

Figure showing that panthera and felis inherit from felidae, tiger inherits from panthera, and housecat inherits from felis.

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.

```
class ChildClass extends ParentClass {
   constructor () {
      super();
      // properties
   }
}
```

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

```
class Felidae {
   constructor() {
      this.claws = "retractable";
   }
}

class Panthera extends Felidae {
   constructor() {
      super();
      this.roar = "loud";
```

```
11    }
12 }
13
14 class Tiger extends Panthera {
15    constructor() {
16       super();
17       this.hasStripes = "true";
18    }
19 }
20
21 let tigger = new Tiger();
22
23 console.log(tigger);
```

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?

1. `Wolf` and `Canis` are parent classes to `Carnivora`.
2. `Wolf` is a child class of `Canis` and a parent class to `Carnivora`.
3. `Wolf` is child class of `Canis`, and `Canis` is a child class of `Carnivora`.
4. `Wolf` is child class of `Canis`, and `Canis` is a parent class of `Carnivora`.

- ← 18.3. Assigning Class Methods
- 18.5. Exercises: Classes →

Back to top

LaunchCode logo

Search

1. Contents
2. 18. Classes
3. 18.5. Exercises: Classes

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. One class should be a `Book` class and two child classes of the `Book` class called `Manual` and `Novel`. Each class will contain two methods. One will be a constructor. The other one will either be in charge of disposal of the book or updating the property related to the number of times a book has been checked out. *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.

Check your solution

2. Declare an object of the `Novel` class for the following tome from the library:

Novel¶

Variable	**Value**
Title	Pride and Prejudice
Author	Jane Austen
Copyright date	1813
ISBN	1111111111111
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:

Manual¶

Variable	Value
Title	Top Secret Shuttle Building Manual
Author	Redacted
Copyright date	2013
ISBN	0000000000000
Number of pages	1147
Number of times the book has been checked out	1
Whether the book has been discarded	No

Check your solution

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.

Check your solution

- ← 18.4. Inheritance
- 18.6. Studio: Classes →

Back to top

LaunchCode logo

Search

1. Contents
2. 18. Classes
3. 18.6. Studio: Classes

18.6. Studio: Classes¶

18.6.1. Getting Started¶

Let's create a class to handle new animal crew candidates!

Edit the practice file as you complete the studio activity.

18.6.2. 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:
    1. Bubba Bear has a mass of 135 kg and test scores of 88, 85, and 90.
    2. Merry Maltese has a mass of 1.5 kg and test scores of 93, 88, and 97.
    3. 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.

18.6.3. 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`.

18.6.4. 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.

18.6.4.1. 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).

18.6.4.2. 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 ___.'`

18.6.5. 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?

- ← 18.5. Exercises: Classes
- 19. Terminal →

Back to top

LaunchCode logo

Search

1. Contents
2. 19. Terminal

19. Terminal¶

- ← 18.6. Studio: Classes
- 19.1. What is a terminal? →

Back to top

LaunchCode logo

Search

1. Contents
2. 19. Terminal
3. 19.1. What is a 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.

Sample graphic user interface.

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.

Sample command line interface.

A CLI with commands navigating the same file paths as the GUI above.¶

The application responsible for running a CLI is called a **terminal** and the program interpreting the commands is called the **shell**.

Note

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

19.1.3. Setup the Terminal¶

Follow these instructions for setting up your terminal. You'll need it for this chapter's exercises and you can explore inside it while you read along with the text.

- ← 19. Terminal
- 19.2. Filesystem and Paths →

Back to top

LaunchCode logo

Search

1. Contents
2. 19. Terminal
3. 19.2. Filesystem and Paths

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**. Folders that exist inside other folders are called **subdirectories**. A **root directory** can refer to a few different things but essentially means the top-most directory of a given system. In other words, a root directory is not a sub-directory - but it will probably contain its own subdirectories. Inside the machine you work with called your computer, the root directory is the location of primary hard drive - in Windows, that's your C drive; in a Mac, the root directory is represented as `/`. The root directory is the **parent directory** for the folders stored inside of it.

Example

Most of you have a `Desktop` folder on your computer. If there is a folder on your Desktop called "LC101_Homework", then the parent directory of `LC101_Homework` is `Desktop`.

A **path** for files and folders is the list of parent directories that the computer must go through to find that particular item.

Filesystems 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 sub, or 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.

Say `homework.js` is in a different directory called `CoderGirl_Homework`. `CoderGirl_Homework` is inside the `Desktop` directory. Your current directory is `LC101_Homework`. In this scenario, 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.

- ← 19.1. What is a terminal?
- 19.3. How to Do Stuff in the Terminal →

Back to top

LaunchCode logo

Search

1. Contents
2. 19. Terminal
3. 19.3. How to Do Stuff in the Terminal

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.

Basic Terminal Commands¶

Command	Result
`ls`	Lists all files and folders in the current directory.
`cd <new-directory>`	`cd` stands for *change directory*. Navigates from the current directory to `new-directory`.
`pwd`	*Print working directory*. Prints the path of the current directory.
`mkdir <new-folder>`	*Make directory*. Creates `new-folder` inside the current directory.
`touch <new-file>`	Creates a file called `new-file` in the current directory.
`rm <old-file>`	*Removes* `old-file` from the current directory.
`man <command>`	*Manual*. Prints to the screen the manual pages for the `command`. This includes the proper syntax and a description of how that command works.
`clear`	Empties the terminal window of previous commands and output.

Command	Result
`cp <source-path> <target-path>`	*Copies* the file or directory at `source-path` and puts it in the `target-path`.
`mv <source-path> <target-path>`	*Moves* the file or directory at `source-path` from its current location to `target-path`.

Note

1. `rm` will permanently remove items from the computer and cannot be undone.
2. Git bash does not support `man`. Instead, `<command> --help` provides a scaled down alternative.

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.

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?

1. prompt
2. command
3. shell
4. There isn't a line that does that.

Question

Which shortcut takes you to the parent directory?

1. .
2. ~
3. . .

- ← 19.2. Filesystem and Paths
- 19.4. Running Programs in the Terminal →

Back to top

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.

- ← 19.3. How to Do Stuff in the Terminal
- 19.5. Exercises: Terminal →

Back to top

LaunchCode logo

Search

1. Contents
2. 19. Terminal
3. 19.5. Exercises: Terminal

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.

Using your terminal, navigate to your Home directory using `cd ~`.
2.
3. Use `ls` to view the contents of your Home directory.

   Check your solution

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?

   Check your solution

6. Use `cd my_first_directory/` to move inside that directory.

7. `pwd` to check your location.

   Check your solution

8. There, make a file called 'my_first_file.txt' with `touch my_first_file.txt`.

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 it's current spot to directly on the Desktop with `cp my_first_file.txt ../my_first_copy.txt`.

    Check your solution

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?

    Check your solution

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.

    Check your solution

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.

    Check your solution

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.

- ← 19.4. Running Programs in the Terminal
- 20. We Built the Internet on HTML →

Back to top

LaunchCode logo

Search

1. Contents
2. 20. We Built the Internet on HTML

20. We Built the Internet on HTML¶

- ← 19.5. Exercises: Terminal
- 20.1. Background →

Back to top

LaunchCode logo

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

Not all browsers handle code the same way. You might notice discrepancies between browsers, such as font or spacing of elements. If you are confident that your code is correct, the discrepancy is likely browser-related.

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. We Built the Internet on HTML
- 20.2. HTML Structure →

Back to top

LaunchCode logo

Search

1. Contents
2. 20. We Built the Internet on HTML
3. 20.2. HTML Structure

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 web page that appears in the tab.

Here is an example of the structure of an HTML page based off of these rules:

```
<!DOCTYPE html>
<html>
   <head>
      <title>My Web Page</title>
      content
   </head>
   <body>
      content
   </body>
</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:

```
<body>
   <!-- This is an important comment -->
</body>
```

Note

Spacing and tabs helps many programmers read through theirs and their colleagues' code. Be aware that doing so in HTML can affect 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`

- ← 20.1. Background
- 20.3. HTML Tags →

Back to top

LaunchCode logo

Search

1. Contents
2. 20. We Built the Internet on HTML
3. 20.3. HTML Tags

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	`<b>`	When surrounding text, makes that text bold.
Emphasis	`<em>`	When surrounding text, makes that text italic.
Hyperlink	`<a>`	Creates hyperlinks.
Image	`<img>`	Denotes images.
Break	` `	A single line break.
Paragraph	`<p>`	Creates a paragraph in text.
Section	`<span>`	Makes a section in text.
Division	`<div>`	Defines an area of the page.
Form	`<form>`	Creates a form for user input.
Unordered List	`<ul>`	Creates an unordered list.
Ordered List	`<ol>`	Creates an ordered list.
List element	`<li>`	

Tag Name	Code	Definition
		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`. `h1` is the top-level heading, `h2` is a sub-heading of an `h1`, and so on. By default, the headings get progressively smaller as the heading level goes up.

A good rule of thumb is to have only one `h1` in a web page and to not skip a level as you add sub-headings. Resist the temptation to use the heading level to change the size of a given heading. The appearance of a heading should be changed using CSS. We will learn how to do this in the next chapter.

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.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Plant-Loving Astronauts</title>
   </head>
   <body>
      <h1>Space Plants Are Cool</h1>
      <p>NASA discovers that plants can live in <b>outer space</b>. Mor
      <!-- add images from NASA of these space plants -->
   </body>
</html>
```

A web page with the heading, Space Plants Are Cool, and the paragraph about NASA's discovery of space plants.

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.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Plant-Loving Astronauts</title>
   </head>
   <body>
      <h1>Space Plants Are Cool</h1>
      <p>NASA discovers that plants can live in <b>outer space</b>. Mor
      <img src = "space-flower.jpg" alt = "Flower floating in space.">
      <!-- This image was taken by NASA and is in the Public Domain -->
   </body>
</html>
```

A web page with the heading, Space Plants Are Cool, and the paragraph about NASA's discovery of space plants with an accompanying picture of a flower floating in space.

The `<img>` 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`

- ← 20.2. HTML Structure
- 20.4. Exercises: HTML →

Back to top

LaunchCode logo

Search

1. Contents
2. 20. We Built the Internet on HTML
3. 20.4. Exercises: HTML

20.4. Exercises: HTML¶

Complete the HTML file for this simple web page. Add lines to `index.html` that do the following.

1. Add a `h1` to the page that says "Why I Love Web Development".

   Check your solution.

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.

   Check your solution.

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.

```
<!DOCTYPE html>
<html>
   <head>
   </head>
   <body>
      <!-- h1 goes here --->
      <!-- ol goes here --->
      <!-- a goes here --->
      <!-- p goes here --->
   </body>
</html>
```

repl.it

Note

repl.it has other HTML inside of the `index.html` file you will be editing. You should only add code, NOT delete!

In addition, when clicking on your link to make sure it works, right-click to open in a new tab.

- ← 20.3. HTML Tags
- 20.5. Studio: Making Headlines →

Back to top

20.5. Studio: Making Headlines¶

20.5.1. Studio Setup¶

In this course, we will mostly use Firefox as our browser. If you haven't installed Firefox, you can get it from Mozilla. Other browsers have good developer tools as well (Chrome, in particular) but we're going to stick with Firefox. If you are more comfortable with another browser, you'll find that most of the developer tools functionality exists in other browsers' dev tools as well.

Firefox's developer tools contain a lot of functionality, and we will only begin to touch on it here, learning more of its functionality later in the unit. In this class, we'll learn about the tools available for working with HTML elements

20.5.2. 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 Firefox'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

The full documentation for Firefox's developer tools covers these items, and much more.

20.5.3. 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:

A screenshot of the New York Times website, with a fake article announcing an astronaut apprenticeship program at LaunchCode.

A Sample Fake Article¶

20.5.3.1. 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.

20.5.4. Resources¶

- Using Firefox's Page Inspector
- Firefox DevTools Documentation

20.5.5. Bonus Mission¶

Try adding your own 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. You can use Dropbox Basic for this!
- 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.

- ← 20.4. Exercises: HTML
- 21. Styling the Web With CSS →

Back to top

LaunchCode logo

Search

1. Contents
2. 21. Styling the Web With CSS

21. Styling the Web With CSS¶

- ← 20.5. Studio: Making Headlines
- 21.1. What is CSS? →

Back to top

LaunchCode logo

Search

1. Contents
2. 21. Styling the Web With CSS
3. 21.1. What is 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. Styling the Web With CSS
- 21.2. CSS Structure →

Back to top

LaunchCode logo

Search

1. Contents
2. 21. Styling the Web With CSS
3. 21.2. CSS Structure

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
}
```

21.2.1.1. 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.

21.2.1.2. 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:

```
selector {
    property: value;
    property: value;
    property: value;
}
```

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, `color`, which governs text 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 `<b>`, `<em>`, and `<span>`) and block display elements do (such as `<h1>`, `<div>`, and `<p>`).

21.2.1.3. 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,

```
p {
   color: pink;
}
```

Using the element selector will make all paragraph elements on the page have pink text.

Class Selector

We can give a few of the paragraphs on the page the class `pink-paragraph` on the HTML document, like so: `<p class="pink-paragraph">content</p>`. If we want to then style the `pink-paragraph` elements, we need to use the class selector in CSS. Here is how our CSS might look:

```
.pink-paragraph {
   color: pink;
}
```

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:

```
#pinkParagraph {
   color: pink;
}
```

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 below:

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.

```
<head>
   <title>My Web Page</title>
   <link rel="stylesheet" type="text/css" href="styles.css">
</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>
```

21.2.2.1. 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

- ← 21.1. What is CSS?
- 21.3. CSS Rules →

Back to top

LaunchCode logo

Search

1. Contents
2. 21. Styling the Web With CSS
3. 21.3. CSS Rules

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
`font-size`	Changes the size of the font.	medium or 20px
`color`	Changes the text color.	black
`font-family`	Changes the font types.	Depends on the browser
`background-color`	Sets the color of the background of an element.	transparent
`text-align`	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>. Mor
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>
```

Made the background color blue, the heading green, and the paragraph text 18 pt. font of the website in the previous chapter about space plants.

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.

- ← 21.2. CSS Structure
- 21.4. Exercises: CSS →

Back to top

LaunchCode logo

Search

1. Contents
2. 21. Styling the Web With CSS
3. 21.4. Exercises: CSS

21.4. Exercises: CSS¶

We have built a website for you to test your CSS knowledge. Check out the repl.it!

For the exercises, add the following style rules to the website:

1. Change the background color to yellow.

   Check your solution.

Change all paragraph text color to green.
2.
3. Change all `h1` to 36 px font size.

   Check your solution.

4. Align all text to the center of the page.

5. Let's say that you don't like aligning all of the text to the center. Use a CSS class to align only the headings to the center of the page.

   Check your solution.

6. Change the font color of elements with the id, `cool-text`, to blue.

7. Use a CSS id to change the elements in the ordered list to a color of your choosing.

   Check your solution.

Note

We learned from the reading that the location of CSS and the selector type can change the order of precedence of the CSS rules. Try different locations and selector types and see what happens!

- ← 21.3. CSS Rules
- 22. Git More Collaboration →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration

22. Git More Collaboration¶

- ← 21.4. Exercises: CSS
- 22.1. What is Git? →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.1. What is Git?

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. Getting Started with Git¶

In order to get started with Git, you need to install Git on your machine and install Visual Studio Code.

22.1.3. Check Your Understanding¶

Question

What is a benefit of using a VCS?

- ← 22. Git More Collaboration
- 22.2. Repositories and Commits →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.2. Repositories and Commits

22.2. Repositories 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? Here's a good rule of thumb:

> Any time a significant change in functionality is made, perform a commit.

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 a longer process than just one command.

The procedure for making a commit to a Git repository includes four 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 -m MESSAGE` creates the new commit with the files that the programmer added, with a message describing the changes included in the commit. Here, `MESSAGE` should be a descriptive message within double-quotes.
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 main
```

```
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 trac

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 <[email protected]>
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 the file exists in the Git repository already, but it 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 to 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.

Tip

It's important to include a descriptive commit message. Such messages are visible in your local Git log, as well as in the commit history on GitHub. A good commit message allows you and your fellow developers to easily identify the changes made in a given commit.

`git log` shows the author of the commit, the date made, the comment, and a 40-character hash. This hash or value is a key for Git to refer to the version. Programmers use these hashes to reference specific commits, or snapshots, in the repository's history.

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`

- ← 22.1. What is Git?
- 22.3. Remote Repositories →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.3. Remote Repositories

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. However, one of the benefits of using a VCS is to store backups. If something bad happens to a coder's machine, they might lose all of the code for their projects! This is where remote repositories come in. Instead of keeping a Git repository only on a local machine, the code base is also saved in a **remote repository**. Any team members working on the project keep copies of the code 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 from 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. Later in this chapter, we will link to an external resource that describes the SSH alternative.

22.3.3. Renaming the Default Branch¶

When we run `git init` to start a new repository, part of the process creates a single, default branch. We can see its name by running the command `git branch`.

```
1 $ git branch
2 * main
```

For the examples in this book, we refer to the default branch as `main`. However, depending on the Git settings on your computer, you might see a different one. This won't affect the performance of your project in any way. That said, GitHub defaults to `main` for all new repositories. To keep our work clear, we should match our local and remote branch names.

Try It!

If your installed version of Git names the default branch something other than `main`, change it with the command:

```
$ git branch -m old-branch-name main
```

In this case, `old-branch-name` becomes `main`.

22.3.4. Contributing to a Remote Repository¶

Once a programmer has a profile on Github and a local copy of a remote repository, they start coding!

After they create a new feature, it is time to make a commit. When working with a remote repo, the commit process includes five steps:

1. `git status`
2. `git add`
3. `git commit`
4. `git push origin main`
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 `main` is the name of the branch that receives the commit.

22.3.5. Check Your Understanding¶

Question

What is the new command for making a commit to a remote repository?

- ← 22.2. Repositories and Commits
- 22.4. Branches →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.4. Branches

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 can work on separate versions of the same website without interfering with each other. Besides collaboration, programmers also use *feature branches* to store and test new ideas for their software.

In the previous section, when checking the status, the top line was `On branch main`. The main branch is the default branch of the repository. Many programmers keep the live version of their code in the main branch. For that reason, major work should be done in a new branch, so it doesn't impact the live software.

Diagram depicting two branches coming off of the main branch.

22.4.2. Creating a New Branch¶

Assume a programmer is on `main` and they want to start building a new feature in a new branch. Their first step is 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?

- ← 22.3. Remote Repositories
- 22.5. Merging in Git →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.5. Merging in Git

22.5. Merging in Git¶

22.5.1. How to Merge¶

A **merge** in Git occurs when two branches are combined in the repository.

Example

Let's say a programmer wants to merge a branch called `test` into `main`. To accomplish this:

1. The programmer switches to the branch they want to merge into. In this case:

```
$ git checkout main
```

2. The programmer then runs the merge command:

```
$ git merge test
```

When successful, the code in the `test` branch is pulled into `main`.

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** occurs 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!

22.5.2.1. 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 them. 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 perform a dry-run and make sure there aren't any conflicts is `git merge --no-commit --no-ff <branch>`. The `--no-commit` and `--no-ff` syntax tells 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. If the programmer wants to retrieve stashed work later, they can do so with the command `git stash pop`.

22.5.3. Check Your Understanding¶

Question

If a programmer is on the branch `test` and wants to merge a branch called `feature` into `main`, what steps should they take?

- ← 22.4. Branches
- 22.6. Exercises: Git →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.6. Exercises: Git

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.

   Check your solution

2. Inside the `Git-Exercises` directory, initialize a new Git repository using `git init`.

3. Use `git branch` to check the default branch name. If necessary, change the branch name to `main`.

4. Add a file called `exercises.txt` using the `touch` command in the terminal.

   Check your solution

5. Commit your local changes using the `git commit` procedures.

6. Add `"Hello World!"` to the file called `exercises.txt`.

7. Commit your local changes following the same steps that you used for step 5.

8. Run the `git log` command. Take a screenshot of the result. Make note of what you see!

   Check your solution

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. Click the *Sign Up* button and follow the on-screen directions.
3. Once you have an account, you are ready to store your remote work.

Before August 13, 2021, when users pushed changes to a remote repository (or pulled new content from it), they were prompted to enter their GitHub username and password. However, a username/password combination is not the most secure method available. This is especially true for people who reuse the same credentials across multiple websites. (You know who you are. Stop doing that!)

In an effort to boost account security, GitHub changed its policy. Users must now create a *Personal Access Token* or an *SSH key* to verify their identity.

Setting up a token for your account is fairly straightforward.

22.6.2.1. Create a Personal Access Token (PAT)¶

To use HTTPS to push and pull from GitHub, users must create a **personal access token**. A PAT takes the place of a password, and the token process is considered more secure than a username/password verification.

Once you create your PAT, you will use it instead of your password to perform HTTPS Git operations. For example, if you are working on a repo with the URL, `https://github.com/username/repo.git`, your terminal might look like the following:

```
$ git clone https://github.com/username/repo.git
Username: your_username
Password: your_token
```

Some users question the need for a PAT, since it looks like another password they have to remember. Rather than diving into a lengthy debate and justification, we'll focus on the main point: *GitHub requires a PAT or similar token*. The platform is incredibly helpful, and we want to use it, so we'll follow their advice.

GitHub provides detailed instructions for setting up your PAT, so we will use their documentation. Follow steps 1 - 9 for Creating a Token carefully.

Tips:

1. The checkboxes in step 7 select the actions you're allowed to perform from the command line (terminal). For now, just choose the *repo* option.

   There's no harm in selecting more options, but you won't need any of them for this course.

2. After you generate your PAT in step 8, *copy and save it somewhere safe*! Your new PAT will NOT be an easy-to-remember sequence of characters (that's the whole point), so you need to record it somewhere.

   If you use a password manager, that's a perfect place to keep your PAT. If you use an unsecured spreadsheet or a folded piece of paper, you want to break that habit now.

3. If you will be pushing and pulling from a repository multiple times in quick succession, you can save your PAT in memory for a short time. Run the command:

   ```
   $ git config credential.helper 'cache --timeout=3600'
   ```

   The next time you access your remote repo, Git will ask for your username and PAT. It will then remember your credentials for a certain amount of time. In the example above, `timeout=3600` saves your information for 1 hour (3600 seconds). You can adjust the amount of time up or down as needed.

4. **Mac Users**: At the bottom of the PAT documentation page, you can find some OPTIONAL instructions for saving your PAT in the MacOS *Keychain* app.

22.6.3. Optional: The SSH Key¶

As an alternative to interacting with GitHub via HTTPS, developers can use the SSH protocol instead. A description of the differences between HTTPS and SSH is beyond the scope of this text. However, we don't need to understand the nuts and bolts of SSH. We just need to be able to use it.

With an SSH key, you can connect to your GitHub repositories without needing to enter your username and PAT each time you push, pull, or perform some other action. This sounds great! The drawback is that it takes more work to set up.

As we mentioned before, this book assumes the HTTPS protocol. However, the GitHub developers make it easy to use either one. If you would like to explore how to create an SSH key, here are the relevant instructions:

1. General info about GitHub and SSH
2. Generate a new SSH key
3. Add the SSH key to your GitHub account
4. Protecting your SSH key

Warning

For each page, make sure you click on the tab that matches your operating system (Mac, Windows, Linux).

- ← 22.5. Merging in Git
- 22.7. Studio: Communication Log →

22.7. Studio: Communication Log¶

Note

If you have not installed Visual Studio Code on your computer yet, now would be the time to do so!

22.7.1. 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 code in tag-team shifts. By the end of the studio, you should have a good idea about how two people can 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. Gitting Ready¶

You are going to simulate a radio conversation between a shuttle pilot and mission control. You and your partner will alternate tasks, so decide who will be the **Pilot** and who will be the **Control**.

Before you and your partner can begin your collaboration, some preparation is required first. You will both start by creating a new repository on your separate GitHub accounts.

22.7.3.1. Step 1: Create a New Local Repository¶

Warning

Be careful if you try to `copy/paste` the `git` commands! The `$` symbol in the sample code represents the terminal prompt. The symbol is NOT part of the commands.

Control and Pilot: Both of you need to complete steps 1 - 6 on your own machines.

1. In the terminal, navigate to your development folder. Enter the following 3 commands to create a new project. Replace `-ROLE` with your part in this studio, either `-pilot` or `-control`.

   ```
   $ mkdir communication-log-ROLE
   $ cd communication-log-ROLE
   $ git init
   ```

   Note

   IMPORTANT: To avoid confusion later, it's critical that you and your partner give different names to your repositories.

   For the remainder of this studio, we will refer to the repo as `communication-log`.

2. Launch Visual Studio Code. Use the *File* menu to open the `communication-log` folder.

3. Create a new file called `index.html` and open it in the workspace.

4. Paste this code into `index.html`:

   ```
   1 <!DOCTYPE html>
   2 <html>
   ```

```
3    <body>
4       <p>Radio check. Pilot, please confirm.</p>
5    </body>
6 </html>
```

5. Save your work, then stage and commit it.

    1. First, check the `status`.

        ```
        $ git status
        On branch main

        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'
        ```

    2. The output shows is that `index.html` is not staged. Let's `add` everything in this directory, then check the `status` again.

        ```
        $ git add .
        $ git status
        On branch main

        Initial commit

        Changes to be committed:
        (use "git rm --cached <file>..." to unstage)

           new file:   index.html
        ```

    3. The output tells us that the file is staged. Now let's `commit`. After that, we can see a record of our progress by using `git log`.

        ```
        $ git commit -m "Started communication log."
        [main (root-commit) e1c1719] Started communication log.
        1 file changed, 5 insertions(+)
        create mode 100644 index.html

        $ git log
        commit 679de772612099c77891d2a3fab12af8db08b651
        Author: Chris <[email protected]>
        Date:   Wed Apr 5 10:55:56 2021 -0500

           Started communication log.
        ```

6. Use the command `git branch` to check the name for the default branch. If necessary, change the name to `main`.

```
$ git branch
* default_name

$ git branch -m default_name main.
```

GitHub uses `main` for its default branch. To make things easier, you should always try to match your local and remote branch names.

Great! You've got your project going locally. The next step is to push it up to GitHub.

22.7.3.2. Step 2: Push Your Repository To GitHub¶

Control and Pilot: Complete steps 1 - 5 on your separate devices and GitHub accounts.

1. Go to your GitHub profile in a web browser. Click on the "+" button to add a new repository (called a *repo* for short).

   The New Repository link in the dropdown menu at top right on GitHub.

   The *New Repository* link is in the dropdown menu at top right on GitHub.¶

2. On the next page, fill in the *Name* and *Description* fields. Also, uncheck the *Initialize this repository with a README* option, then click *Create Repository*.

   Creating a new repository in GitHub by filling out the form.

   Create a new repository in GitHub.¶

   Note

   If you initialize with a README, Git will refuse to merge the remote repo with your local one. There are ways around this, but it's faster and easier to just create an empty repo on GitHub.

3. After clicking, you should see something similar to:

   The page you see after creating an empty repository, with several options.

   Connecting to a repository in GitHub.¶

4. Now go back to your terminal and copy/paste the commands shown in the GitHub instructions. These should be very similar to:

   ```
   $ git remote add origin https://github.com/your-username/communicatior
   $ git branch -M main
   $ git push -u origin main
   ```

   Note

The first time you push up to GitHub, you will be prompted to enter your account username and personal access token. Do this.

You will then see a large amount of output that you can safely ignore. The final few lines will confirm a successful push. They will look something like this:

```
To github.com:your-username/communication-log.git
   c7f97814..54993de3  main -> main
```

Warning

Unless you've set up an SSH key with GitHub, make sure you've selected the HTTPS option in the Quick Setup. If you're not sure whether you have an SSH key, you probably don't.

5. Confirm that GitHub has the same version as your local project. Click around and see what is there. You can view all your code through GitHub's web interface. The files and code you see in your browser should match what you have in Visual Studio Code!

   A repository with one commit in GitHub

   A repository with one commit in GitHub.¶

22.7.4. Git the Teamwork Started!¶

You've successfully created a new GitHub repository and pushed content to it. Now it's time for you and your partner to start collaborating on the same repo.

For the remaining sections of this studio, keep an eye on the *Control* and *Pilot* role tags. Make sure that you both perform your tasks in the recommended order. Mixing things up won't destroy the universe, but it will make finishing the studio more complicated.

Even when it is not your turn to complete a task, read and observe what your partner is doing. The steps here mimic a real-world collaborative Git workflow.

22.7.4.1. Step 3: Add A Collaborator¶

Control, the first step is yours. In order for **Pilot** to make changes to your GitHub repository, you must invite them to collaborate.

1. **Control**: In your web browser, go to your `communication-log` repo. Click the *Settings* button then select the *Manage Access* option.

   Click "Settings" and "Manage Access" to let other users modify the repo.

   Manage access to your repo.¶

2. **Control**: Click on the green *Invite a collaborator* button. Enter your partner's GitHub username and click *Add to repository*.

   Enter a GitHub username, then click the Add button.

   Choose who else can modify your GitHub repo.¶

3. **Pilot**: You should receive an email invitation to join this repository. View and accept the invitation.

Note

Pilot: If you don't see the email, check your Spam folder. If you still don't have the email, login to your GitHub account. Visit the URL for Control's copy of the repo. You should see an invite notification at the top of the page.

22.7.4.2. Step 4: Clone Project from GitHub¶

Warning

Pilot, did you and your partner give different names to your `communication-log` repositories?

If not, take a moment to find your *local* `communication-log` folder on your machine. RENAME IT!

1. **Pilot**: Go to Control's GitHub profile and find their `communication-log` repo. Click on the green *Code* button. Select the HTTPS option and copy the URL to your clipboard.

   The Code button is on the right-hand side of a project's main page.

   Cloning a repository in GitHub¶

2. **Pilot**: In your terminal, navigate back to your development folder and clone Control's repo. You should be OUTSIDE of any other Git repositories.

   The clone command looks something like this:

   ```
   $ git clone https://github.com/username/communication-log.git
   ```

   Replace the URL with the address you copied from GitHub.

3. **Pilot**: You should now have a copy of **Control's** project on your machine.

22.7.5. Git Talking¶

Whew! That was quite the setup experience. Now you're ready to dive into the main part of the assignment.

On to Studio Part 2!

- ← 22.6. Exercises: Git
- 22.8. Studio: Communication Log (cont.) →

Back to top

LaunchCode logo

Search

1. Contents
2. 22. Git More Collaboration
3. 22.8. Studio: Communication Log (cont.)

22.8. Studio: Communication Log (cont.)¶

Tip

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 this in our Git Stash tutorial.

22.8.1. Step 5: First Message Exchange¶

1. **Pilot**: Use the *File* menu in Visual Studio Code to open the cloned `communication-log-control` directory. Double click the `index.html` file to open it in the editor.

   Modify the HTML to add your response to mission control. Be creative, the communication can go anywhere! Just don't ask your partner what you should write.

2. **Pilot**: After you finish, commit your change with the usual `git status/ git add ./git commit -m` process.

3. **Pilot**: Push up your changes up to GitHub so **Control** can see them as well. Use the command:

   ```
   $ git push origin main
   Counting objects: 9, done.
   remote: Resolving deltas: 100% (8/8), completed with 8 local objects.
   To [email protected]:username/communication-log.git
      511239a..679de77  main -> main
   ```

4. **Control**: Pull Pilot's changes down from GitHub with the command

```
$ git pull origin main
From github.com:username/communication-log
   e0de62d..e851b7e  main     -> origin/main
Updating e0de62d..e851b7e
Fast-forward
index.html | 1 +
1 file changed, 1 insertion(+)
```

5. **Control**: Notice that the code your local `index.html` file changes to reflect the line(s) Pilot added. Cool!

   Respond by adding a new HTML element and some text. Save, commit, and push your changes up to GitHub.

6. **Pilot and Control**: Play with the `pull/edit/push` process for a while! Repeat the cycle a few more times to add to your story.

Tip

In VS Code, right-click on the `index.html` tab. Choose the *Copy Path* option.

[Menu options that appear after right-clicking a file tab in VS Code. "Copy Path" is highlighted.]

Next, open a web browser and paste the path into the address bar. Ta da! Your webpage appears. Opening `index.html` in your browser lets you track your progress by refreshing the page.

Notice that the path in the address bar looks very similar to the result we would see from the `pwd` command in the terminal.

22.8.2. Step 6: Create a Branch In Git¶

This back-and-forth workflow is nice, but it can get in the way. After all, professional developers don't sit around waiting for their teammates to commit and push a change before starting their own work. Fortunately, Git branches allow partners to work on a project at the same time and at their own pace.

1. **Pilot**: While Control is working on the next part of the story, use the terminal to create a new branch called `open-mic`. Recall that a *branch* is a separate copy of the codebase. This lets you commit changes without affecting the code in the `main` branch.

   ```
   $ git checkout -b open-mic
   Switched to a new branch 'open-mic'

   $ git branch
     main
   * open-mic
   ```

2. **Pilot**: In VS Code, create a new file called `style.css` and add the following CSS style rule:

```
body {
   color: white;
   background-color: black;
}
```

3. **Pilot**: Next, open `index.html` and link to the stylesheet. Your HTML should look something like this:

```
<!DOCTYPE html>
<html>
   <head>
      <link rel="stylesheet" type="text/css" href="style.css">
   </head>
   <body>
      <p>Radio check. Pilot, please confirm.</p>
      <!-- Your ongoing conversation here... -->
   </body>
</html>
```

4. **Pilot**: Save and commit your changes, then push the new branch up to GitHub with the command:

   ```
   $ git push origin open-mic
   ```

   Note that the command is a little different than what you used before (`git push origin main`). The final part is the name of the branch that you want to push to GitHub.

5. **Control**: To pull down the new `open-mic` branch, enter the command:

   ```
   $ git pull origin open-mic
   ```

   Note

   If you don't know the name of the new branch, then just enter `git pull`. This will pull down *all* new branches and show you a list of the options.

6. **Pilot and Control**: If you haven't already done so, open `index.html` in a browser to see what it looks like.

Good! Now let's take a look at GitHub and find the new `open-mic` branch.

22.8.2.1. View Branches in GitHub¶

Pilot and Control: To view the available branches on GitHub, select *Branches* from the navigation section just below the repository title.

A GitHub repository page with the Branches button highlighted.

The branches Button in GitHub¶

Great progress! Now let's figure out how to merge two branches in GitHub.

22.8.3. Step 7: Open a Pull Request in GitHub¶

1. **Pilot**: In your browser, go to the GitHub project and click on *Branches* and make sure you see the new branch name, *open-mic*.

   The Branches page of a repo, with a button to open a new pull request to the right of each feature branch.

   The branches page in GitHub.¶

2. **Pilot**: Click *New Pull Request* to ask Control to review your changes in the `open-mic` branch before merging them into `main`. Add some text in the description box to let Control know what you did and why.

   Note

   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*.

   The form for creating a new pull request.

   Open a PR in GitHub.¶

This is what an opened pull request looks like:

An open pull request.

An open PR in GitHub¶

22.8.4. Step 8: Merge the Pull Request¶

1. **Control**: Go to your repo in GitHub. Click on *Pull Requests*. Next, click on the title for the one and only PR.

   Review the PR details.

   PR Open in GitHub¶

2. **Control**: The page that opens shows the history of all the commits made to the `open-mic` branch. When ready, click the green *Merge Pull Request* button, followed by *Confirm Merge*.

   Confirm the merge request.

   Finally! Merge the pull request.¶

3. Upon a successful merge, you should see a screen similar to the following:

   The screen displayed after a PR is merged

   PR Merged in GitHub¶

4. **Pilot and Control**: The changes from `open-mic` are now in the `main` branch, but only in the remote repository on GitHub. You will need to pull the updates to your `main` for them to be present locally.

   ```
   $ git checkout main
   $ git pull origin main
   ```

   Git is able to merge these files on its own.

Notice that the changes made in the `open-mic` branch now appear in `main`. Refreshing the tab in your browser should display the styled webpage!

Tip

When you save a change to our HTML code, clicking the *Refresh* button in the browser displays the new layout. However, this doesn't always work for changes made to the CSS. Browsers often save the stylesheet in memory to speed up reloading. If the browser continues to use the old code, you won't see your new styles.

To fix this, force a clean reload of the page. For most browsers (like Firefox, Chrome, and Safari), hold down the `Shift` key and click *Refresh*. For Microsoft Edge, use the `Control` key plus *Refresh*.

22.8.5. Resources¶

Before moving on to the bonus studio content, here are some external resources for using Git and GitHub.

1. Git Branching - Basic Branching and Merging
2. Adding Another Person To Your Repository
3. Resolving Conflicts In the Command Line

22.8.6. Bonus: Merge Conflicts!¶

When teaming up on a project, things won't always go smoothly. It's common for two people to change the same line(s) of code on their separate machines. This prevents Git from being able to automatically finish a merge.

An animated GIF file showing two opposing armies colliding in a mess.

Merge conflicts!¶

Merge conflicts often occur, and they are not a big deal. To see how to handle this situation, you will intentionally create a merge conflict and then resolve it.

1. **Pilot**: In VS Code, switch back to the `main` branch.

2. **Pilot**: Change the `style.css` file. The webpage is looking pretty plain, so spice up the `body` style rule 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:

   Our HTML page with a fancy font.

   Satisfying!¶

3. **Pilot**: Save and commit the changes, then push them up to GitHub.

   ```
   $ git push origin main
   ```

Meanwhile...

1. **Control**: In VS Code, switch back to the `main` branch.

2. **Control**: In your local `style.css` file, change the `body` rule 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 }
   ```

3. **Control**: Save and commit your changes to `main`.

22.8.6.1. Resolve the Merge Conflicts¶

1. **Control**: Try to push your changes up to GitHub. You should get an error message. How exciting!

   ```
   $ git push origin main

   To [email protected]:username/communication-log.git
   ! [rejected]        main -> main (fetch first)
   error: failed to push some refs to '[email protected]:username/communi
   hint: Updates were rejected because the remote contains work that you
   hint: not have locally. This is usually caused by another repository p
   hint: to the same ref. You may want to first integrate the remote char
   hint: (e.g., 'git pull ...') before pushing again.

   There's a lot of text in the message. However, the main idea is clear:
   ``Updates were rejected because the remote contains work that you do r
   locally.``
   ```

2. Somebody (**Pilot**, in this case), pushed changes to `main`, and you don't have those commits on your computer. To fix this, begin by pulling those changes down from GitHub:

   ```
   $ git pull

   From github.com:username/communication-log
      7d7e42e..0c21659  main     -> origin/main
   Auto-merging style.css
   CONFLICT (content): Merge conflict in style.css
   Automatic merge failed; fix conflicts and then commit the result.
   ```

   Since **Pilot** and **Control** both made changes to the same lines of code, Git cannot automatically merge the changes.

3. **Control**: The specific locations where Git could not automatically merge files are indicated by lines that begin with `CONFLICT`. You will have to edit these files yourself to incorporate Pilot's changes. Open the `style.css` file.

   VS Code shows merge conflicts in the editor window

   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).

4. **Control**: To fix the conflicts, you need to tell Git what code to keep. You can use the provided buttons to either `Accept Current Change` (which

is your code), `Accept Incoming Change` (from **Pilot**), or you can update the highlighted statements directly in the editor. We'll let you decide how to handle this!

5. **Control**: Save, commit, and push your changes. You should see no error message this time.

6. **Pilot**: Pull down the new updates. Notice that since **Control** dealt with the merge conflicts, you don't have to!

Congratulations! You just fixed your first merge conflict!

22.8.6.2. More Merge Conflicts!¶

Turn the tables so **Pilot** can practice resolving a merge conflict.

1. **Control and Pilot**: Decide which file and lines of code you will both change. Make *different* changes in those places.
2. **Control**: Save, commit, and push your changes up to GitHub.
3. **Pilot**: Try to pull down the changes, and notice that there are merge conflicts. Resolve them, then save, commit, and push the result.
4. **Control**: Pull down the final, resolved code.

22.8.7. Avoiding Conflicts¶

Git happens. Merge conflicts will pop up, but they're not a big deal. Still, the best way to handle them is to try to avoid them in the first place. Here are some tips:

1. Deal with any uncommitted work before trying to merge.
2. Partners should avoid working on the same file at the same time.
3. Try to avoid adding code directly into `main`. New ideas should be explored in a different branch first and then merged.

- ← 22.7. Studio: Communication Log
- 23. The DOM and Events →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events

23. The DOM and Events¶

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.1. JavaScript and the Browser

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 its 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. Since each browser uses its engine, each browser may handle HTML, CSS, or JavaScript differently. This can lead to discrepancies between browsers.

Warning

The website Can I Use is a great resource to check browser usability of any JavaScript, HTML or CSS.

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.

23.1.2.1. 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.

23.1.2.2. Inline JavaScript¶

Example

Notice the <script> tag below contains JavaScript code that will be executed by the browser.

```
<!DOCTYPE html>
<html>
<head>
      <title>Embedded JavaScript Example</title>
      <script>
         // JavaScript code goes here!
         console.log("Hello from inside the web page!");
      </script>
</head>
<body>
      contents
</body>
</html>
```

Console Output

```
Hello from inside the web page!
```

23.1.2.3. 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.

```
<!DOCTYPE html>
   <html>
   <head>
      <title>External JavaScript Example</title>
      <script src = "myjs.js"></script>
   </head>
   <body>
      <!-- content -->
   </body>
   </html>
```

Then the JavaScript file, `myjs.js` might look something like this.

```
1 // JavaScript code goes here!
2 console.log("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. The DOM and Events
- 23.2. The DOM →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.2. The DOM

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.

Figure showing the tree-like relationship between the document of the DOM and HTML elements on the 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.

```
<!DOCTYPE html>
<html>
<head>
      <title>Using DOM Variables</title>
      <script>
         console.log("the page title:", document.title);
         console.log("the protocol:", window.location.protocol);
      </script>
</head>
<body>
      contents
</body>
</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.

```
<!DOCTYPE html>
<html>
<head>
      <title>Add content using DOM</title>
</head>
<body>
      <p id="main-text">Words about things...</p>
      <script>
         let p = document.getElementById("main-text");
         p.append("More words about things");
         console.log(p.innerHTML);
      </script>
</body>
</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?

1. Word documents you have downloaded
2. Directives of memory
3. The browser window, HTML document, and the elements

Question

What is the value of `p.innerHTML`?

```
<p id="demo-text">Hello friend</p>
<script>
   let p = document.getElementById("demo-text");
```

```
4    console.log(p.innerHTML);
5 </script>
```


- ← 23.1. JavaScript and the Browser
- 23.3. More DOM Methods and Properties →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.3. More DOM Methods and Properties

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.

Window Properties and Methods¶

Method or Property	Syntax	Description
alert	`window.alert("String message")`	Displays a dialog box with a message and an "ok" button to close the box.
confirm	`window.confirm("String message")`	Displays a dialog box with a message and returns `true` if user clicks "ok" and `false` if user clicks "cancel".
location	`window.location`	Object that represents and alters the web address of the window or tab.

Method or Property	Syntax	Description
console	`window.console`	Represents the debugging console. Most common and basic use is `window.console.log()`.

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.

Document Properties and Methods¶

Method or Property	Syntax	Description
title	`document.title`	Read or set the title of the document.
getElementById	`document.getElementById("example-id")`	Returns a reference to the element that's `id` attribute matches the given string value.
querySelector	`document.querySelector("css selector")`	Returns the first element that matches the given CSS selector.
querySelectorAll	`document.querySelectorAll("css selector")`	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.

Element Properties and Methods¶

Method or Property	Syntax	Description
getAttribute	`element.getAttribute("id")`	Returns the value of the attribute.
setAttribute	`element.setAttribute("id", "string-value")`	Sets the attribute to the given value.
style	`element.style.color`	Object that allows reading and setting *INLINE* CSS properties.
innerHTML	`element.innerHTML`	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 = window.confirm("String message");
```

Question

Which of these are TRUE about selecting DOM elements?

1. You can select elements by *CSS class* name
2. You can select elements by *id attribute* value
3. You can select elements by *tag* name
4. All of the above

- ← 23.2. The DOM
- 23.4. Events →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.4. Events

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

```
<!DOCTYPE html>
<html>
<head>
      <title>Button click handler</title>
</head>
<body>
      <button onclick="console.log('you rang...');">Ring Bell</button>
</body>
</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.

```
<!DOCTYPE html>
<html>
<head>
      <title>Button click handler</title>
   <script>
         function youRang() {
            document.getElementById("main-text").innerHTML += "you rang
            console.log("you rang...");
         }
```

```
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?

1. Holding an event
2. Having an event
3. Handling an event

- ← 23.3. More DOM Methods and Properties
- 23.5. Event Listeners →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.5. Event Listeners

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.

```
anElement.addEventListener("eventName", aFunction);
```

`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"`.

```
<!DOCTYPE html>
<html>
<head>
   <title>Use addEventListener</title>
</head>
<body>
   <p id="main-text" class="orange" style="font-weight: bold;">
      a bunch of really valuable text...
   </p>
   <button id="ring-button">Ring Bell</button>
   <script>
      function youRang() {
         document.getElementById("main-text").innerHTML += "you rang...
         console.log("you rang...");
      }
      // Obtain a reference to the button element
      let button = document.getElementById("ring-button");
```

```
18      // Set named function youRang as the click event handler
19      button.addEventListener("click", youRang);
20   </script>
21 </body>
22 </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 usablity.

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 anElement.addEventListener("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 can be passed to the event handler function. This event is an object instance of the Event class, which defines methods and properties related to events.

```
1 anElement.addEventListener("eventName", function(event) {
2   console.log("event type", event.type);
3   console.log("event target", 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.

```
<!DOCTYPE html>
<html>
<head>
   <title>Event Bubbling</title>
   <style>
      #toolbar {
          padding: 20px;
          border: 1px solid black;
          background-color:darkcyan;
      }
  </style>
</head>
<body>
   <div id="toolbar">
      <button id="ring-button">Ring Bell</button>
   </div>
   <script>
      let button = document.getElementById("ring-button");
      button.addEventListener("click", function (event) {
          console.log("button clicked");
      });
      document.getElementById("toolbar").addEventListener("click", func
          console.log("toolbar clicked");
      });
```

```
25      document.addEventListener("click", function (event) {
26          console.log("document clicked");
27      });
28   </script>
29 </body>
30 </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 button.addEventListener("click", function (event) {
2   console.log("button clicked");
3   event.stopPropagation();
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?

- ← 23.4. Events
- 23.6. Event Types →

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.

```
<!DOCTYPE html>
<html>
<head>
   <title>Window Load Event</title>
   <script>
      // add load event handler to window
      window.addEventListener("load", function() {
          // put DOM code here to ensure elements have been loaded
          console.log('window loaded');

          let ring = document.getElementById("ring-button");
          ring.addEventListener("click", function (event) {
```

```
            console.log("ring ring");
        });

        let knock = document.getElementById("knock-button");
        knock.addEventListener("click", function (event) {
            console.log("knock knock");
        });
    });
  </script>
</head>
<body>
  <div id="toolbar">
    <button id="ring-button">Ring Bell</button>
    <button id="knock-button">Knock on Door</button>
  </div>
</body>
</html>
```

Console Output (if "Knock on Door" button is clicked)

```
window loaded
knock knock
```

23.6.2. `mouseover` and `mouseout` Events¶

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 cursor enters an element.

Example

We can use `mouseover` event to add a `">"` to the `innerHTML` of the element that the mouse cursor has been moved over.

```
<!DOCTYPE html>
<html>
<head>
  <title>Mouseover Event</title>
  <script>
      window.addEventListener("load", function() {
        let list = document.getElementById("lane-list");
        list.addEventListener("mouseover", function (event) {
          let element = event.target;
          element.innerHTML += ">";
          console.log("target", element);
        });
      });
  </script>
</head>
<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>>>>>>>>
```

Similarly, there is a **mouseout event** that is triggered when the cursor leaves a given element.

23.6.3. Check Your Understanding¶

Question

Which error occurs 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`?

1. The console is clear.
2. All elements and resources have been loaded by the browser.
3. Your files have finished uploading.

- ← 23.5. Event Listeners
- 23.7. Exercises: The DOM and Events →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.7. Exercises: The DOM and Events

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 liftoff!`. The *Take off* button has an `id="liftoffButton"` attribute.

   Check your solution

2. When the user's cursor goes over the *Abort Mission* button, the button's background turns red. The *Abort Mission* button has `id="abortMission"`.

3. When the user's cursor *leaves* the *Abort Mission* button, the button's background returns to its original state (*Hint:* Setting the background color to the empty string, `""`, will force it to revert to the default browser styles.)

   Check your solution

4. When the user clicks the *Abort Mission* button, make a confirmation window that says `Are you sure you want to abort the mission?`. If the user confirms that they want to abort, the text changes to `Mission aborted! Space shuttle returning home`.

- ← 23.6. Event Types
- 23.8. Studio: The DOM and Events →

Back to top

LaunchCode logo

Search

1. Contents
2. 23. The DOM and Events
3. 23.8. Studio: The DOM and Events

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.

23.8.1. Getting Started¶

First, fork the studio repository to your Github account. To do so, on the studio repository page on Github, click the "Fork" button.

Image showing the "Fork" button is in the top right on the repository page.

A popup appears asking where to fork the repository to and you select your profile. You should now have a copy of the repository on your own profile!

Note

Not only is forking repositories an important Git skill, it is especially vital in the class so that everyone has the same starter code! Before continuing, make sure that the repository is now on your profile and you are working with your copy of the starter code for the rest of the studio.

If you have properly forked a repository, when you click on the forked repository on your profile, you will see the following:

Image showing that the repository name says "Forked from" in the top left when the repository has been forked.

Once you have properly forked the repository, you can clone the remote repository to your computer. To start, click on the green "Clone" button to get the proper HTTPS url for the command.

Image showing that a popup appears once the "Clone" button is clicked with the proper url.

Copy the url either by clicking on the button with a clipboard icon or highlighting the whole url and copying.

In the terminal, navigate to the directory where you want to put your new project. Use the command, `git clone <url>`, with the url you just copied to put the project on your local machine.

Note

`git clone` will clone a whole directory including the Git repository on your machine, so there is no need to initialize a Git repository in a new directory to get started this way!

Open Visual Studio Code and go to *File* > *Open* to find your new project and get started!

Warning

When you push your work up to your Github repository, you may notice the option to create a pull request. Please do NOT do so. This pull request will be linking your work back the shared starter code. If you accidentally do so, you can navigate to the starter code repository page and find your submission under "Pull Requests". Click on your submission and close the pull request at the bottom of the page.

23.8.2. The Requirements¶

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 parts b-d.
    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.

    Note

    When you are moving the shuttle, you want to use absolute positioning in CSS. Absolute positioning means positioning the object based on its location in the parent object. In the case of our flight simulator, the parent object is a div with the id, `shuttleBackground`. Relative positioning means positioning the object based on its fellow child objects. We might use relative positioning if there were planet objects within our `shuttleBackground` div.

    When setting the position of an object in CSS, you use a string that ends in "px". So the position of 10 pixels is "10px". To add a number of pixels to the position, you may first have to use `parseInt` to convert the current position to a number.

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 parts b-d.
    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 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.

23.8.3. 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.

- ← 23.7. Exercises: The DOM and Events
- 24. HTTP: The Postal Service of the Internet →

Back to top

LaunchCode logo

Search

1. Contents
2. 24. HTTP: The Postal Service of the Internet

24. HTTP: The Postal Service of the Internet¶

Back to top

LaunchCode logo

Search

1. Contents
2. 24. HTTP: The Postal Service of the Internet
3. 24.1. How the Internet Works

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.

A server surrounded by several client computers, connected by arrows.

Several clients communicating with a server.¶

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.

Common Web Protocols¶

Protocol	Full Name	Role
HTTP	Hypertext Transfer Protocol	High-level web communication for transferring files and information, including: • 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 *packets*
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.

24.1.2.1. 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.

24.1.2.2. 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.

24.1.2.3. 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.

24.1.3.1. 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 `://`.

24.1.3.2. 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`.

24.1.3.3. 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.

24.1.3.4. 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-2080d6e23ef41463553f203daaa15991f
```

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.

24.1.3.5. 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. HTTP: The Postal Service of the Internet
- 24.2. HTTP at a Glance →

Back to top

LaunchCode logo

Search

1. Contents
2. 24. HTTP: The Postal Service of the Internet
3. 24.2. HTTP at a Glance

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.

A client and a server exchanging HTTP requests and responses

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.

An addressed envelope

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.

An image of a mail processing facility next to an image of network cables.

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¶

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.

24.2.3. Check Your Understanding¶

Question

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.

- ← 24.1. How the Internet Works
- 24.3. Requests →

Back to top

LaunchCode logo

Search

1. Contents
2. 24. HTTP: The Postal Service of the Internet
3. 24.3. Requests

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/2
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`.

24.3.1.1. 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

24.3.1.2. 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.

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 `POST` 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

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.)

- ← 24.2. HTTP at a Glance
- 24.4. Responses →

Back to top

LaunchCode logo

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:

A generic 404 message displayed in a browser.

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.

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 `Location` 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.

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, responses 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?

- ← 24.3. Requests
- 24.5. HTTP in the Browser →

Back to top

LaunchCode logo

Search

1. Contents
2. 24. HTTP: The Postal Service of the Internet
3. 24.5. HTTP in the Browser

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:

Firefox's developer tools, with an empty Network pane.

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:

Firefox's developer tools, with several requests in the Network pane.

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.

The details of an HTTP request, viewed in the Network pane.

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.

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?

- ← 24.4. Responses
- 25. User Input with Forms →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms

25. User Input with Forms¶

- ← 24.5. HTTP in the Browser
- 25.1. Forms →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms
3. 25.1. 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.

```
<html>
   <head>
      <title>Form Example</title>
   </head>
   <body>
      <!-- empty form -->
      <form></form>
   </body>
</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.

```
<html>
   <head>
      <title>Form Example</title>
   </head>
   <body>
      <form>
         <input type="text"/>
      </form>
   </body>
</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.

```
<html>
   <head>
      <title>Form Example</title>
   </head>
   <body>
      <form>
         <label>Username <input type="text" name="username"/></label>
         <label>Team Name <input type="text" name="team"/></label>
      </form>
   </body>
</html>
```

HTML that includes a form tag with two input elements. Each element is inside of a label element.

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>`.

```
<label for="username">Username</label>
<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

Try clicking on the `Username` and `Team Name` labels below. What happens?

Username Team Name

```
<form>
   <label>Username <input type="text" name="username"/></label>
   <label>Team Name <input type="text" name="team"/></label>
</form>
```

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.

Subscribe to Newsletter

```
<form>
   <label>Subscribe to Newsletter
      <input type="checkbox" name="newsletter"/>
   </label>
</form>
```

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.

Language

```
<form>
   <label>Language <input name="language" type="text" value="JavaScript"
</form>
```

25.1.5. Check Your Understanding¶

Question

What is the purpose of the `name` attribute for `input` elements?

Question

Which `input` attribute sets the default value?

- ← 25. User Input with Forms
- 25.2. Form Submission →

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.

```
<form>
   <label>Username <input type="text" name="username"></label>
   <!-- clicking either of these will cause a form submission -->
   <input type="submit"/>
   <button>Submit</button>
</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.

```
<html>
   <head>
      <title>Form Example</title>
      <style>
         body { padding: 25px;}
      </style>
   </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

Browser screen shot showing form with two text inputs and a submit button. Both inputs have text values.

Output After Submitted

Browser screen shot showing form after it has been submitted. The URL has queryString showing.

Run it.

Notice in the above example that the browser address has changed to:

```
https://form-default--launchcode.repl.co/?username=salina&team=Space+Coder
```

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, we will soon learn how to submit form data via POST.

Note

Since spaces are not allowed in URLs, the browser replaces them with +.

25.2.1.1. 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?

- ← 25.1. Forms
- 25.3. POST Form Submission →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms
3. 25.3. POST Form Submission

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"
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

Screen shot of firefox browser with form loaded and network tab open.

Firefox browser with form loaded, network tab open, and Persist logs checked¶

1. Enter data into the inputs
    - Type `tracking` into Username input
    - Type `Requests` into Team Name input
2. Click Submit button

Web page showing submitted values and the entries in network tab.

Firefox browser with request handler loaded and network tab showing requests¶

1. Inspect the data sent in the POST request

POST request highlighted with Params tab open showing Form data.

POST request highlighted with Params tab open showing Form data¶

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?

Question

What attribute on `<form>` determines *where* the request is sent?

- ← 25.2. Form Submission
- 25.4. Text Inputs →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms
3. 25.4. Text Inputs

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"
   <label>Code Name<input type="text" name="codeName"/></label>
   <label>Code Word<input type="password" name="codeWord"/></label>

   <!-- textarea must have open and closing tags -->
   <label>Mission Description<br/>
      <textarea name="description" rows="5" cols="75"></textarea>
   </label>

   <button>Send Report</button>
</form>
```

Form with Code Name, Code Word, and Description field. All fields have values.

Submitted Values

```
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

`Run it <https://repl.it/@launchcode/basic-inputs-example>`__
```

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?

- ← 25.3. POST Form Submission
- 25.5. Specialized Text Inputs →

Back to top

LaunchCode logo

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	`<input type="date" name="flightDate"/>`	Browser validates the value is a valid date format. Some browsers provide a *date picker*.	
email	`<input type="email" name="emailAddress"/>`	Browser validates the value is a valid email address format.	
number	`<input type="number" name="fuelTemp"/>`	Browser validates the value is a valid number format.	

25.5.1. Example¶

Example

```
<form action="https://handlers.education.launchcode.org/request-parrot" me
  <label>Email<input type="email" name="emailAddress"/></label>
  <label>Report Date<input type="date" name="reportDate"/></label>
  <label>Crew Count<input type="number"
  name="crewCount" min="1" max="10"/></label>
  <button>Send Report</button>
</form>
```

Form with Code Name, Code Word, and Description field. All fields have values.

Submitted Values

```
[email protected]
reportDate=2019-03-08
crewCount=8
```

Run it

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 *checked*.	sign up

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 at repl.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"/></labe
```

```
<label>Butter<input type="checkbox" name="ingredient" value="butter"/></la
<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?

- ← 25.5. Specialized Text Inputs
- 25.7. Radio Input →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms
3. 25.7. Radio Input

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	`<input type="radio" name="crewReady" value="yes"/>`	A small circle that allows selecting *one* of multiple values. Used in groups of two or more.	yesno

25.7.1. Example¶

Example

```
<form action="https://handlers.education.launchcode.org/request-parrot" me
   Flight Rating:
   <label>Rough<input type="radio" name="flightRating" value="rough"/></la
   <label>Few Bumps<input type="radio" name="flightRating" value="fewBumps
   <label>Smooth<input type="radio" name="flightRating" value="smooth"/></
```

```
   <button>Send Report</button>
</form>
```

Form for flight rating, with a radio inputs for rough, few bumps, and smooth.

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?

- ← 25.6. Checkbox Input
- 25.8. Select Input →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms
3. 25.8. Select Input

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	`<select name="weather"><option value="1">clear</option><option value="2">cloudy</option></select>`	A menu that allows selection of one option. Requires options to be in `<option>` tags.	

25.8.1. Example¶

Example

```
<form action="https://handlers.education.launchcode.org/request-parrot" me
   <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

Web form showing select inputs for Operation Code and Facility.

Select "Rocket Test" and "White Sands Test Facility, NM"

Web form with select inputs with "Rocket Test" and "White Sands Test Facility, NM" selected.

Submitted Values

```
operation=2
facility=white-sands
```

Run it

25.8.2. Check Your Understanding¶

Question

For a select input, what determines the value that is submitted during form submission?

- ← 25.7. Radio Input
- 25.9. Validation with JavaScript →

Back to top

LaunchCode logo

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.

```
<!DOCTYPE html>
<html>
<head>
      <title>Check input value with DOM</title>
</head>
<body>
   <form>
      <label>Language
         <input type="text" name="language" id="language" value="JavaSc
      </label>
   </form>
   <button id="update">Update Input Value</button>
   <script>
      let button = document.getElementById("update");
      // add event handler for when button clicked
      button.addEventListener("click", function() {
         let input = document.getElementById("language");
         console.log(input.value);
         // now update the value in the input
```

```
20          input.value = input.value + " rocks!";
21          console.log(input.value);
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
    1. If the values are valid, allow the form submission
    2. 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       window.addEventListener("load", function() {
11          let form = document.querySelector("form");
12          form.addEventListener("submit", function(event) {
13             alert("submit clicked");
14          });
15       });
16    </script>
17    <body>
```

```
18      <form method="POST" action="https://handlers.education.launchcode
19         <label>Username <input type="text" name="username"></label>
20         <label>Team Name <input type="text" name="team"></label>
21         <button>Submit</button>
22      </form>
23   </body>
24 </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   window.addEventListener("load", function() {
3      let form = document.querySelector("form");
4      form.addEventListener("submit", function(event) {
5         let usernameInput = document.querySelector("input[name=usernam
6         // alert the current value found in the username input
7         alert("username: " + usernameInput.value);
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   window.addEventListener("load", function() {
3      let form = document.querySelector("form");
4      form.addEventListener("submit", function(event) {
5         let usernameInput = document.querySelector("input[name=usernam
6         let teamName = document.querySelector("input[name=team]");
7         if (usernameInput.value === "" || teamName.value === "") {
8            alert("All fields are required!");
9         }
```

```
    });
  });
</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.

```
<script>
  window.addEventListener("load", function() {
    let form = document.querySelector("form");
    form.addEventListener("submit", function(event) {
      let usernameInput = document.querySelector("input[name=usernam
      let teamName = document.querySelector("input[name=team]");
      if (usernameInput.value === "" || teamName.value === "") {
        alert("All fields are required!");
        // stop the form submission
        event.preventDefault();
      }
    });
  });
</script>
```

25.9.4. Check Your Understanding¶

Question

What method on the `event` object can be used to stop a form submission?

- ← 25.8. Select Input
- 25.10. Exercises: Forms →

Back to top

LaunchCode logo

Search

1. Contents

25.10. Exercises: Forms¶

Hello programmer, we need you to make a Rocket Simulation form.

25.10.1. Form Data¶

This is the kind of data the Rocket Simulation form will need to process.

Data Fields for the Rocket Simulation Form¶

Display Name	Input Type	Input Name	Possible Values
Test Name	text	`testName`	No limitations
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

25.10.2. Form Display¶

Your completed simulation form will look roughly like this:

Rocket simulation form with all input fields filled out.

Submitted Values

```
testName=Moon+Shot
testDate=2020-07-16
rocketType=Lynx
boosterCount=3
windRating=10
productionServers=on
```

25.10.3. Starting Codebase¶

Code your solution in this repl.it.

25.10.4. Instructions¶

Please follow the steps below and good luck!

1. Create a `<form>` with these attributes.
    1. Set `method` to `"POST"`
    2. Set `action` to `"https://handlers.education.launchcode.org/request-parrot"`

    Check your solution.

2. Add a `<label>` and `<input>` for Test Name to the `<form>`.
    1. `<label>Test Name <input type="text" name="testName"/></label>`.

3. Can you submit the form now? What is missing?

    Check your solution.

4. Add a `<button>Run Simulation</button>` to the `<form>`.

5. Enter a value into the "testName" input and submit the form.
    1. Was the value properly submitted to the form handler?

    Check your solution.

6. Repeat steps 2 and 5 for the remaining data fields from the data table.
    1. Pay attention to the input types and possible options.
    2. Don't forget to add a `<label>` for each input.

25.10.5. 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.

- ← 25.9. Validation with JavaScript
- 25.11. Studio: HTTP and Forms →

Back to top

LaunchCode logo

Search

1. Contents
2. 25. User Input with Forms
3. 25.11. Studio: HTTP and Forms

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:

A form with a text input and radio buttons corresponding to various search engines.

The search engine selector that we will build.¶

Most search engines work the same way. The 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¶

1. Go to the studio repository and fork to your Github account.
2. Clone the repository and `cd` into the new directory.

25.11.3. Create Form Inputs¶

Let's build out the form in `index.html`. We will need some data for the search engines we want to work with.

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

1. Create a text input within the form and set its `name` attribute to the value `"q"`.
2. 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.
3. Create a `label` element for each radio button.
4. Finally, add a submit button to the form and set it's `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.

25.11.4.1. 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 window.addEventListener('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.

25.11.4.2. 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:

```
let actions = {
   "google": "https://www.google.com/"
};
```

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.

- ← 25.10. Exercises: Forms
- 26. JSON →

Back to top

LaunchCode logo

Search

1. Contents
2. 26. JSON

26. JSON¶

- ← 25.11. Studio: HTTP and Forms
- 26.1. Introduction →

Back to top

LaunchCode logo

Search

1. Contents
2. 26. JSON
3. 26.1. Introduction

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. Over the course of the next two chapters, 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)** 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.

We interact with computers using various interfaces, either with a GUI or a CLI. However, an application does not know how to use a GUI, or a CLI, and needs its own interface to communicate with another application. An API is the interface that allows one application to communicate with another application.

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. JSON
- 26.2. Data Formats and JSON →

Back to top

LaunchCode logo

Search

1. Contents
2. 26. JSON
3. 26.2. Data Formats and JSON

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".

```
{
    "title": "An Astronaut's Guide to Life on Earth",
    "author": "Chris Hadfield",
    "ISBN": 9780316253017,
    "year_published": 2013,
    "subject": ["Hadfield, Chris", "Astronauts", "Biography"],
    "available": true
}
```

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 it's 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, 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".

```
{
    "hits": 3,
    "book": [
        {
            "title": "An Astronaut's Guide to Life on Earth",
             "author": "Chris Hadfield",
             "ISBN": 9780316253017,
             "year_published": 2013,
             "subject": ["Hadfield, Chris", "Astronauts", "Biography"],
             "available": true
        },
        {
            "title": "Astronaut",
            "author": "Lucy M. George",
            "ISBN": 9781609929411,
            "year_published": 2016,
            "subject": ["Astronauts", "Juvenile Fiction", "Space statio
            "available": false
        },
```

```
        {
            "title": "Astronaut Ellen Ochoa",
            "author": "Heather E. Schwartz",
            "ISBN": 9781512434491,
            "year_published": 2018,
            "subject": ["Ochoa Ellen", "Women astronauts", "Astronauts"
            "available": true
        }
    ]
}
```

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:

```
{
    "title": "The Cat in the Hat",
    "author": "Dr. Seuss"
}
```

JavaScript object:

```
let newBook = {
    title: "The Cat in the Hat",
    author: "Dr. Seuss"
}
```

To represent a string in JSON, you MUST use double quotes. In JavaScript, you can use double quotes or single quotes.

JSON:

```
{
    "title": "The Last Astronaut",
    "author": "David Wellington"
}
```

JavaScript object:

```
let anotherBook = {
    title: 'The Last Astronaut',
    author: 'David Wellington'
}
```

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

True or False: JSON is JavaScript.

Question

What purpose does JSON serve?

- ← 26.1. Introduction
- 26.3. JSON →

Back to top

LaunchCode logo

Search

1. Contents
2. 26. JSON
3. 26.3. JSON

26.3. JSON¶

1. Which of the following three code snippets is correct JSON syntax? Why are the other two options incorrect?

    1. ```
       {
           type: "dog",
           name: "Bernie",
           age: 3
       }
       ```

    2. ```
       {
           "type": "dog",
           "name": "Bernie",
           "age": 3
       }
       ```

    3. ```
       {
           "type": 'dog',
           "name": 'Bernie',
           "age": 3
       }
       ```

    Check your solution.

2. Which of the following three code snippets is correct JSON? Why are the other two options incorrect?

    1. ```
       {
           "animals": [
              {
                    "type": "dog",
                    "name": "Bernie",
                    "age": 3
              },
              {
                    "type": "cat",
                    "name": "Draco",
                    "age": 2
              }
           ]
       }
       ```

    2. ```
       {
           [
              {
                    "type": "dog",
                    "name": "Bernie",
                    "age": 3
              },
              {
                    "type": "cat",
       ```
       ```

```
                "name": "Draco",
                "age": 2
            }
        ]
    }
```

3.
```
[
    {
        "type": "dog",
        "name": "Bernie",
        "age": 3
    },
    {
        "type": "cat",
        "name": "Draco",
        "age": 2
    }
]
```


- ← 26.2. Data Formats and JSON
- 27. Fetch →

Back to top

LaunchCode logo

Search

1. Contents
2. 27. Fetch

27. Fetch¶

- ← 26.3. JSON
- 27.1. Fetching Data →

Back to top

LaunchCode logo

Search

1. Contents
2. 27. Fetch
3. 27.1. Fetching Data

27.1. 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 web page. The URL for this special LaunchCode weather API is https://handlers.education.launchcode.org/static/weather.json.

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
11       </div>
12    </body>
13 </html>
```

Warning

Before going through the `fetch` examples, please know that `fetch` does NOT work in Internet Explorer. List of alternative browsers

27.1.1. `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.
    1. For this example, it will be `"https://handlers.education.launchcode.org/static/weather.json"`
2. A response handler function to utilize the data that is being fetched.
    1. For this example, it will be `function(response){...};`

Example

Notice a string URL is passed to `fetch`. Also notice the anonymous *request handler function* that has a `response` parameter. The `.then` method will be explained soon.

```
fetch("https://handlers.education.launchcode.org/static/weather.json").the
   console.log(response);
} );
```

In this example, we are requesting data from `https://handlers.education.launchcode.org/static/weather.json` and our response handler (the anonymous function) simply logs the response to the console.

27.1.2. `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 6.
2. A `fetch` and *response handler function* on line 7.

3. A `console.log(response);` on line 8 that prints out the `response` object.

```
 1           <!DOCTYPE html>
 2 <html>
 3    <head>
 4       <title>Launch Status</title>
 5       <script>
 6          window.addEventListener("load", function() {
 7             fetch("https://handlers.education.launchcode.org/static/wea
 8                console.log(response);
 9             } );
10          });
11       </script>
12    </head>
13    <body>
14       <h1>Launch Status</h1>
15       <h3>Weather Conditions</h3>
16       <div id="weather-conditions">
17          <!-- TODO: dynamically add html about weather using data from
18       </div>
19    </body>
20 </html>
```

repl.it

Let's break down how `fetch` works:

1. A URL is passed to `fetch` as a parameter.
2. This causes an HTTP GET request to be sent from the browser to the API.
3. The API processes the request and sends back a response, which contains data.
4. Once the browser receives the response, the `.then()` statement executes.
5. The anonymous *response handler function* called by `.then(function(response)` runs, and this function deals with the data sent back from the API.
6. Based on the code in the handler function, the web page gets 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.

27.1.2.1. 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.

Screen shot showing developer tools open with the network call to the API highlighted.

The GET request to the Weather API highlighted in developer tools.¶

In the image above, 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.

27.1.2.2. Response Object¶

The response to the GET request is contained in a `Response` object, which is an instance of the Response class. The class contains methods that allow us to access the status of an API request and the data returned in the response.

Example

On line 8, the `json()` method is used to gain access to the JSON data contained in the `response` object.

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          window.addEventListener("load", function() {
 6             fetch("https://handlers.education.launchcode.org/static/wea
 7                // Access the JSON in the response
 8                response.json().then( function(json) {
 9                   console.log(json);
10                });
11             });
12          });
13       </script>
14    </head>
15    <body>
16       <h1>Launch Status</h1>
17       <h3>Weather Conditions</h3>
18       <div id="weather-conditions">
19          <!-- TODO: dynamically add html about weather using data from
```

```
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
}
```

27.1.2.3. 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 display that data.

Example

1. On line 8, the `div` object is defined and linked to the HTML element with the id `weather-conditions`.
2. On line 10, the `innerHTML` property of the `div` object is set to be the HTML elements in lines 11 - 16.

```
 1 <html>
 2    <head>
 3       <title>Launch Status</title>
 4       <script>
 5          window.addEventListener("load", function() {
 6             fetch("https://handlers.education.launchcode.org/static/wea
 7                response.json().then( function(json) {
 8                   const div = document.getElementById("weather-conditio
 9                   // Add HTML that includes the JSON data
10                   div.innerHTML = `
11                      <ul>
12                         <li>Temp ${json.temp}</li>
13                         <li>Wind Speed ${json.windSpeed}</li>
14                         <li>Status ${json.status}</li>
15                         <li>Chance of Precip ${json.chanceOfPrecipitati
16                      </ul>
17                   `;
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

Let's take a look at the expected sequence of events:

1. In line 6, the `fetch` command sends a request to the URL.
2. When the API returns a response, the `then` method calls the anonymous handler function and passes in the `response` object (also line 6).
3. On line 7, the handler function tries to retrieve the JSON data from the `response` object. When this is successful, the anonymous `function(json)` gets called.
    1. On line 8, the `div` object is defined and linked to the HTML element with the id `weather-conditions`.
    2. On line 10, the `innerHTML` property of the `div` object is set to be the HTML elements in lines 11 - 16.
    3. The HTML in lines 11 - 16 gets filled in using weather data stored in the `json` object.
4. The two anonymous functions end, and the HTML defined in lines 11 - 16 gets added to the `div` element on line 26.

Opening the developer tools on the web page shows the added HTML:

Screen shot of browser showing Launch Status web page with the weather data in HTML.

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`.

27.1.3. Check Your Understanding¶

Question

What is the correct syntax for `fetch`?

1. `fetch("GET", "https://api.url").then(...);`
2. `fetch("https://api.url").doStuff(...);`
3. `fetch("https://api.url").then(...);`

- ← 27. Fetch
- 27.2. Asynchronous and Promises →

Back to top

LaunchCode logo

Search

1. Contents
2. 27. Fetch
3. 27.2. Asynchronous and Promises

27.2. 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.

27.2.1. 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.

27.2.2. 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 fetch("https://handlers.education.launchcode.org/static/weather.json").t
2    console.log(response);
3 } );
```

To make it clearer, let's capture the value returned by `fetch` in a variable named `fetchPromise`.

```
1 const fetchPromise = fetch("https://handlers.education.launchcode.org/st
2 fetchPromise.then( function(response) {
3    console.log(response);
4 } );
```

`fetch` returns an instance of the `Promise` class. The `Promise` class represents a **promise**. A promise is the *eventual* outcome of an asynchronous event. In the above example, `fetchPromise` represents the eventual response from the HTTP request to `https://handlers.education.launchcode.org/static/weather.json`.

A promise can be fulfilled or rejected. When a promise is fulfilled, data is passed to the response handler function. The `then` method of `Promise` defines what will happen when the promise is fulfilled. When a promise is rejected, the error reason is returned.

The above example can be translated to these steps

1. Make an HTTP request to `https://handlers.education.launchcode.org/static/weather.json`
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

27.2.3. More Promises¶

Above, we showed a promise representing the outcome of an HTTP request, however, promises can represent the outcome of *any* asynchronous event. For example, 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 example below shows how promises are used to represent two different types of asynchronous events and the outcomes.

Example

```
1 const fetchPromise = fetch("https://handlers.education.launchcode.org/st
2 fetchPromise.then( function(response) {
3    const jsonPromise = response.json();
4    jsonPromise.then( function(json) {
5       console.log("temp", json.temp);
6    });
7 } );
```

This example involves two promises. On line 1, `fetchPromise` is a promise that represents the fetch request. On line 3, `jsonPromise` is a promise that represents the response data being turned into JSON.

Finally on line 5, the JSON data can be logged.

Tip

Promises can be a hard concept to understand. Focus on the examples and the theory will make sense in time.

27.2.4. Check Your Understanding¶

Question

We know exactly when an asynchronous request will return.

1. True
2. False

Question

A promise can represent the outcome of *any* future event.

1. True
2. False

Question

`then` is a method of the `Promise` class that allows us to run code after an event is completed.

1. True
2. False

- ← 27.1. Fetching Data
- 27.3. Exercises →

Back to top

LaunchCode logo

27.3. Exercises¶

27.3.1. Fetch¶

To practice fetching data, create a file called `fetch_planets.html` using `touch` in your terminal.

Add this preliminary HTML to your `fetch_planets` document:

```
<!DOCTYPE html>
<html>
   <head>
      <title>Fetch Planets</title>
      <script>
         window.addEventListener("load", function(){
            // TODO: fetch planets JSON
         });
      </script>
   </head>
   <body>
      <h1>Destination</h1>
      <div id="destination">
         <h3>Planet</h3>
      </div>
   </body>
</html>
```

1. The URL where our planet data is located is: `"https://handlers.education.launchcode.org/static/planets.json"`. Add the code to fetch this URL inside the load event listener.

   Check your solution.

2. Peek at the `response` returned in the request by adding a print statement inside of the function.

   Copy the file path of your HTML file and paste it as the URL in your browser. You won't see much on the page yet. Open your developer tools and examine both the *Console* tab for the response value, as well as the *Network* tab for the request status.

3. Use the `.json()` method on your response now to see more of the data in the console: What data type do you see printed?

   Check your solution.

4. Replace your `console.log(json)` with the following to view a portion of the JSON into the app.

```
const destination = document.getElementById("destination");
destination.innerHTML = `<h3>Planet ${json[0].name}</h3>`;
```

   Refresh the page to see some new data in your HTML. Play around by changing the index number. Does the planet name change? Can you change the planet's property being printed?

5. Now, what happens if we move those last lines we added to outside and after the fetch request?

   Since `json` hasn't been defined outside of the `response.json()` method yet, in order to move the template literal that uses that `json` variable, we'll need to initiate it outside of the function call. Let's also put our print statement back so we can verify that our fetch works.

   Refresh the page and try this. See any data? See anything of note in the console?

   Check your solution.

6. Our last task left us with some knowledge about where and how we can use the fetched data, but we don't really want to keep those changes. Instead, how about we use an event to change the planet information we see? Let's move the DOM manipulation to inside a click handler.

```
fetch("https://handlers.education.launchcode.org/static/planets.jso
   response.json().then(function(json) {
      const destination = document.getElementById("destination");
      destination.addEventListener("click", function(){
         destination.innerHTML = `
            <div>
               <h3>Planet ${json[0].name}</h3>
               <img src=${json[0].image} height=250></img>
            </div>
         `;
      });
   });
});
```

Now, after refreshing the page, you can click on the `Planet` header to make the name and image appear. Take note, we're still fetching on load, just not displaying the data until the the header is clicked.

7. For fun and good measure, let's dynamically change which planet's info we're displaying each time the header is clicked. To do this,

    1. Declare a counter variable, `index` that changes each time a `click` event takes place.
    2. Use the value of `index` as the position in the planets array to use in the template literal.
    3. Finally, since we want to cap the value of `index` so that it does not exceed the length of the planets array, use a modulo to control how large `index` can get.

    Et voila! Our destination changes on each click!

    Check your solution.

    Clicking through destinations.

    Put on your planetary shoes. We are moving through planets!

- ← 27.2. Asynchronous and Promises
- 27.4. Studio: Fetch & JSON →

Back to top

LaunchCode logo

Search

1. Contents
2. 27. Fetch
3. 27.4. Studio: Fetch & JSON

27.4. Studio: Fetch & JSON¶

Your task is to build a website that shows astronauts fetched from an API.

27.4.1. Get Started¶

1. Fork the studio repository to your own Github account.
2. Clone the repository to your computer.

27.4.2. Requirements¶

1. Add code that runs on the `window` `load` event.
    1. 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://handlers.education.launchcode.org/static/astronauts.json
    1. Do this part inside the `load` event handler.
3. Add each astronaut returned to the web page.
    1. Use the HTML template shown below.
    2. Be sure to use the exact HTML including the CSS classes. (starter code contains CSS definitions)

27.4.2.1. 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.

```
[
   {
      "id": 1,
      "active": false,
      "firstName": "Mae",
      "lastName": "Jemison",
      "skills": [
            "Physician", "Chemical Engineer"
      ],
      "hoursInSpace": 190,
      "picture": "mae-jemison.jpg"
   },
   {
      "id": 2,
      "active": false,
      "firstName": "Frederick",
      "lastName": "Gregory",
      "skills": [
            "Information Systems", "Shuttle Pilot", "Fighter Pilot", "H
      ],
      "hoursInSpace": 455,
      "picture": "frederick-gregory.jpg"
   },
   {
      "id": 3,
      "active": false,
      "firstName": "Ellen",
      "lastName": "Ochoa",
      "skills": [
            "Physics", "Electrical Engineer"
      ],
      "hoursInSpace": 979,
      "picture": "ellen-ochoa.jpg"
   },
   {
      "id": 4,
      "active": false,
      "firstName": "Guion",
```

```
      "lastName": "Bluford",
      "skills": [
            "Aerospace Engineer", "Philosophy", "Physics", "Colonel USA
            "Fighter Pilot"
      ],
      "hoursInSpace": 686,
      "picture": "guion-bluford.jpg"
   },
   {
      "id": 5,
      "active": false,
      "firstName": "Sally",
      "lastName": "Ride",
      "skills": [
            "Physicist", "Astrophysics"
      ],
      "hoursInSpace": 343,
      "picture": "sally-ride.jpg"
   },
   {
      "id": 6,
      "active": true,
      "firstName": "Kjell",
      "lastName": "Lindgren",
      "skills": [
            "Physician", "Surgeon", "Emergency Medicine"
      ],
      "hoursInSpace": 15,
      "picture": "kjell-lindgren.jpg"
   },
   {
      "id": 7,
      "active": true,
      "firstName": "Jeanette",
      "lastName": "Epps",
      "skills": [
            "Physicist", "Philosophy", "Aerospace Engineer"
      ],
      "hoursInSpace": 0,
      "picture": "jeanette-epps.jpg"
   }
]
```

27.4.2.2. HTML Template¶

Create HTML in this exact format for each astronaut, but include data about that specific astronaut. For example the HTML below 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.

```
<div class="astronaut">
   <div class="bio">
      <h3>Mae Jemison</h3>
      <ul>
         <li>Hours in space: 190</li>
         <li>Active: false</li>
         <li>Skills: Physician, Chemical Engineer</li>
      </ul>
   </div>
   <img class="avatar" src="images/mae-jemison.jpg">
</div>
```

27.4.2.3. Expected Results¶

After your code loads the data and builds the HTML, the web page should look like:

Screen shot showing what result of studio should look like.

Example of what resulting page should look like.¶

27.4.3. Bonus Missions¶

1. Display the astronauts sorted from most to least time in space.
2. Make the "Active: true" text green, for astronauts that are still active (active is true).
3. Add a count of astronauts to the page.

- ← 27.3. Exercises
- 28. TypeScript →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript

28. TypeScript¶

Back to top

28.1. 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.

28.1.1. Set Up Your Local Development Environment¶

As we continue to shift away from online work in repl.it to *local* development on your computer, you need to install some specific software.

1. If you have not already done so, download and install Visual Studio Code on your machine.
2. Install Node on your computer, which also installs the NPM Command Line Interface (CLI). We discussed this briefly in the Modules chapter.

The NPM CLI tool provides you with an efficient and deliberate way to install other software and modules onto your computer. As you proceed through the next four chapters, you will use the NPM CLI in the terminal to perform these installations.

- ← 28. TypeScript
- 28.2. Declaring and Using Variables →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript
3. 28.2. Declaring and Using Variables

28.2. 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 variableName: type = value;
```

28.2.1. `number`¶

When declaring a variable and using the `number` type, we add `number` to the variable declaration, like so:

```
let variableName: number = 10;
```

28.2.2. `string`¶

When declaring a `string`, we want to use the `string` keyword.

```
let variableName: string = "10";
```

28.2.3. `boolean`¶

The `boolean` keyword should be used when declaring a variable of the `boolean` type.

```
let variableName: boolean = true;
```

28.2.4. 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.

```
// In JavaScript, we have:

let spaceShuttleName = "Determination";
let shuttleSpeed = 17500;
let distancetoMars = 225000000;
let distancetoMoon = 384400;
let milesperKilometer = 0.621;

// The same declarations in TypeScript would be:

```

```
let spaceShuttleName: string = "Determination";
let shuttleSpeed: number = 17500;
let distancetoMars: number = 225000000;
let distancetoMoon: number = 384400;
let milesperKilometer: number = 0.621;
```

28.2.5. Check Your Understanding¶

Question

The correct declaration of the variable, `astronautName`, that holds the value, `"Sally Ride"`, is:

1. `let astronautName = "Sally Ride";`
2. `let astronautName = string: "Sally Ride";`
3. `let astronautName: string = "Sally Ride";`
4. `string astronautName = "Sally Ride";`

- ← 28.1. Why TypeScript?
- 28.3. Arrays in TypeScript →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript
3. 28.3. Arrays in TypeScript

28.3. Arrays in TypeScript¶

Arrays in TypeScript must contain values of the same type. When declaring an array, the type needs to be declared.

```
let arrayName: number[] = [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.

```
let tupleName: [number, string, number];

tupleName = [10, "9", 8];
```

28.3.1. Examples¶

In JavaScript, we would declare an array holding items in our cargo hold like so:

```
let cargoHold = ['oxygen tanks', 'space suits', 'parrot', 'instruction man
```

In TypeScript, we would declare the same `cargoHold` array a little differently:

```
let cargoHold: string[] = ['oxygen tanks', 'space suits', 'parrot', 'instr
```

What about declaring arrays for elements on the Periodic Table? In JavaScript, that is a relatively simple task:

```
1 let element1 = ['hydrogen', 'H', 1.008];
2 let element2 = ['helium', 'He', 4.003];
3 let element26 = ['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 element1: [string, string, number];
2 element1 = ['hydrogen', 'H', 1.008];
3
4 let element2: [string, string, number];
5 element2 = ['helium', 'He', 4.003];
6
7 let element26: [string, string, number];
8 element26 = ['iron', 'Fe', 55.85];
```

28.3.2. Check Your Understanding¶

Question

Which of the following statements is FALSE about a tuple in TypeScript?

1. Tuples can hold as many elements as needed.
2. Tuples hold values of one type.
3. When declaring a tuple, programmers include the types of the values in a tuple.

- ← 28.2. Declaring and Using Variables
- 28.4. Functions in TypeScript →

Back to top

LaunchCode logo

28.4. Functions in TypeScript¶

When creating functions in TypeScript, we have many of the same options as in JavaScript. We can make anonymous functions, give them a different number of parameters, and so on. However, when working with functions in TypeScript, we have to keep *data types* in mind.

28.4.1. Declaring Functions¶

28.4.1.1. Named Functions¶

Let's take a look at a TypeScript function declaration.

Example

We want to declare a function called `myFunction`. `myFunction` has one parameter, a number called `x`. `myFunction` returns the value of `x` multiplied by 2.

With this in mind, the declaration in TypeScript would look like:

```
function myFunction(x: number): number {
   return x*2;
}
```

Here you can see that we provided the type of the parameter and the type of the value that is returned after the colons. If `x` were a string or a boolean, then we would replace `number` with the data type of `x`.

What if the function doesn't return a value? In these cases, we use `void` as the return type.

Example

Let's change up `myFunction` a little bit! `myFunction` still has one parameter, our number called `x`, however, it doesn't return a specific value.

```
let y: number = 0;

function myFunction(x: number): void {
   y = x*2;
}
```

Instead of returning the value of x multiplied by 2, myFunction now assigns the value of x multiplied by 2 to another variable, y. We can now use void to specify that no value is returned.

28.4.1.2. Anonymous Functions¶

For an anonymous function, you still need to provide types for the value returned and the parameters.

Example

Now we want to declare an anonymous function, myFunction, which has one parameter, a number called x, and returns the value of x multiplied by 2.

```
let myFunction = function(x: number): number { return x*2; };
```

Just as we did above with the named function, we need to make sure that we include the data type for the parameters and the return type of the function.

28.4.2. Optional Parameter¶

When declaring a function in TypeScript, you may make some of your parameters optional. This means that when you are calling the function, you can leave off the optional parameter(s).

To denote a parameter as optional, we use the ? notation. Any parameters that are optional must follow the required parameters.

Example

```
1     function myFunction(a: number, b?:number): number {
2             if (typeof b !== 'undefined'){
3                     return a+b+5;
4                } else {
5                       return a+5;
6                }
7          }
8
9 console.log(myFunction(1,2));
10 console.log(myFunction(1));
11 console.log(myFunction(3,5));
12 console.log(myFunction(3));
```

```
8
6
13
8
```

In this example, the myFunction function has two parameters, a and b. a is required and b is optional. When both arguments are provided, then the sum

of the 2 arguments and 5 is returned. When only one argument is provided to the function, then the value of `a+5` is returned.

Another way to deal with a parameter you need to be optional is to give it a default value.

Let's say that in the example above, we wanted to give `b` a default value of 5. That way, if no argument is supplied for `b`, then the value returned is that of `a+10`.

Example

```
function myFunction(a: number, b = 5): number {
   return a+b+5;
}

console.log(myFunction(1,2));
console.log(myFunction(1));
console.log(myFunction(3,5));
console.log(myFunction(3));
```

```
8
11
13
13
```

Because `b` has a default value of `5`, when the user does not pass a value to the second argument of `myFunction`, 5 is used as the value of `b`.

28.4.3. Check Your Understanding¶

Question

What is wrong with this function declaration? NOTE there are at least 3 things that should be changed.

```
let myFunction = function(a:number,b? = 3) {return a*b};
```


- ← 28.3. Arrays in TypeScript
- 28.5. Classes and Interfaces in TypeScript →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript
3. 28.5. Classes and Interfaces in TypeScript

28.5. Classes and Interfaces in TypeScript¶

28.5.1. Classes¶

Classes in TypeScript look something like this:

```
 1 class Astronaut {
 2    name: string;
 3    constructor(firstName: string, lastName: string) {
 4       this.name = firstName + " " + lastName;
 5    }
 6    greet() {
 7       return "Hello, " + this.name;
 8    }
 9 }
10
11 let Bob = new Astronaut("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 Panthera {
 2    roar: string;
 3    constructor(currentRoar: string) {
 4       this.roar = currentRoar;
 5    }
 6 }
 7
 8 class Tiger extends Panthera {
 9    stripes: boolean = true;
10
11 }
12
13 let tigger = new Tiger("loud");
14 console.log(tigger.roar);
15 console.log(tigger.stripes);
```

28.5.2. Interfaces¶

Interfaces are not used in JavaScript, but are important in TypeScript. 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.

Note

Though the use of interfaces in Angular is not within the scope of this book, interfaces are used rather frequently in Angular code and are important in object-oriented programming languages, such as Java.

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.

```
interface Astronaut {
   name: string;
}

function astronautName (astronaut: Astronaut): string {
   return astronaut.name;
}

let bob = {name: "Bob"};
console.log(astronautName(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.

```
interface interfaceName {
   someProperty: number;
}

class className implements interfaceName {
   constructor(x: number) {
      this.someProperty = x;
   }
}
```

Example

```
interface Panthera {
   roar: string;
}

class Tiger implements Panthera {
   roar: string;

   constructor() {
      this.roar = 'rooooaaaarrrr';
   }
}

class Lion implements Panthera {
   roar: string;

   constructor() {
      this.roar = 'ROOOOAAAAARRRRRR';
   }
}

function pantheraSounds(panthera: Panthera): void {
   console.log(`Panthera says ${panthera.roar}`);
}

let tiger = new Tiger();
let lion = new Lion();

pantheraSounds(tiger);
pantheraSounds(lion);
```

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` interface.

28.5.2.1. 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:

```
let giraffeTwo = {
     species: "Reticulated Giraffe",
     name: "Alicia",
     weight: null,
     age: 10,
     diet: "leaves"
};
```

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.

```
interface giraffeTwo = {
     species: string;
     name: string;
     weight?: number;
     age: number;
     diet: string;
};
```

28.5.3. `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:

```
export class className {
   // properties and methods
}
```

28.5.4. `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:

```
import { className } from 'relativefilepath';

let newClass = new className;
```

28.5.5. Check Your Understanding¶

Question

What is the difference between a class and an interface?

- ← 28.4. Functions in TypeScript
- 28.6. Compiling TypeScript →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript
3. 28.6. Compiling TypeScript

28.6. Compiling TypeScript¶

To work with TypeScript in our Node environment, we'll need to install the TypeScript module. We talked before about running JavaScript in Node. Unlike JavaScript, TypeScript cannot be run directly in Node. TypeScript files must first be transformed, or compiled, into JavaScript.

In this section, you will install the compiler and then practice compiling and running a small TypeScript program from the terminal.

To install TypeScript, in your terminal, type

```
$ npm install -g typescript
```

Tip

Mac users: You may receive a permissions error when running this install command. If you do, try running it as sudo.

Once you have TypeScript installed, you'll be able to run

```
$ tsc -v
```

`tsc` is the TypeScript compiler and `-v` is the version flag to show us that we have TypeScript and it's compiler installed.

28.6.1. `hello_world.ts`¶

In true tutorial form, let's write a quick `Hello, World` program so we can see this compiling in action.

In your code editor, create a new file called `hello_world.ts`. In it, initialize a variable called `message` with the string value `'Hello, World'`. Print the message.

```
1 let message: string = 'Hello, World';
2 console.log(message);
```

Make sure to save your `hello_world.ts` changes before continuing forward!

In the terminal, navigate into the directory that houses your `hello_world.ts` file. Compile the code with the following command:

```
$ cd Desktop/
$ ls
hello_world.ts
$ tsc hello_world.ts
$ ls
hello_world.js hello_world.ts
```

If your TypeScript file is free of syntax errors, you won't see a response from the terminal directly. However, you should see that you now have a newly generated file in your working directory called `hello_world.js`. Open it up and take a look!

Back in the terminal, run that JavaScript file

```
$ node hello_world.js
Hello, World
```

Look at that! Your message printed. You're now running your own code straight from the terminal!

Note

Changing your TypeScript file won't update the corresponding JavaScript file unless you re-compile.

You'll need these compiling and executing steps for the exercises in the following section, so refer back here as needed.

- ← 28.5. Classes and Interfaces in TypeScript
- 28.7. Exercises: TypeScript →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript
3. 28.7. Exercises: TypeScript

28.7. Exercises: TypeScript¶

28.7.1. Part 0 - Get the Starter Code¶

1. Login to your GitHub account.
2. Fork the typescript-lc101-projects repository.
3. Use the terminal to clone your fork from GitHub. If you need a reminder for how to do this, refer to the Git studio.
4. Use the terminal to navigate into the `typescript-lc101-projects` folder, then into the `exercises` subfolder.

```
$ ls
   typescript-lc101-projects
$ cd typescript-lc101-projects
$ ls
   exercises       studio
$ cd exercises
$ ls
   SpaceLocation.ts     parts1-5.ts    tsconfig.json
```

28.7.2. Part 1 - Declare Variables With Type¶

Run VSCode and open the `typescript-lc101-projects` folder. From the file tree, select the `parts1-5.ts` file.

VSCode file tree for the TypeScript exercises.

VSCode file tree¶

In the space indicated, declare and assign a variable for each of the following:

Variable Name	Type	Value
spacecraftName	string	`'Determination'`
speedMph	number	17500
kilometersToMars	number	225000000
kilometersToTheMoon	number	384400
milesPerKilometer	number	0.621

Check your solution

28.7.3. Part 2 - Print Days to Mars¶

In the *same* file you opened in Part 1, do the following.

1. Declare and assign these variables.
    1. Remember: variable declarations in TypeScript include the type!
    2. `milesToMars` is a number with the value of `kilometersToMars * milesPerKilometer`.
    3. `hoursToMars` is a number with the value of `milesToMars / speedMph`.
    4. `daysToMars` is a number with the value of `hoursToMars / 24`.
2. Write a `console.log` statement that prints out the days to Mars.
    1. Use template literal syntax and the variables `${spacecraftName}` and `${daysToMars}`.
3. Use the terminal in VSCode to compile your `.ts` file, then use the command `node parts1-5.js` to run the JavaScript file created during the build process.

Terminal

```
$ tsc parts1-5.ts
$ node parts1-5.js
   Determination would take 332.67857142857144 days to get to Mars.
```

Check your solution

28.7.4. Part 3 - Create a Function¶

1. In the space indicated, define a function that calculates the days it would take to travel to a location.
    1. Function name `getDaysToLocation`
    2. Parameter
        - `kilometersAway` must be a number.
    3. Returns the number of days to a location.
        - Use the same calculations as in Part 2.1.
        - Inside the function, make the variable names generic. Use `milesAway` and `hoursToLocation` instead of `milesToMars` and `hoursToMars`.
        - The function should declare that it returns a `number`.
2. Print out the days to Mars.
    1. Move the output statement from part 2 below your new function.
    2. Refactor the template literal to use `${getDaysToLocation(kilometersToMars)}` and `${spacecraftName}`.
3. Print out the days to the Moon.
    1. Add another output statement and template literal using `${getDaysToLocation(kilometersToTheMoon)}` and `${spacecraftName}`.
4. Use the terminal in VSCode to recompile your `.ts` file, then run the `parts1-5.js` file again.

Terminal

```
$ tsc parts1-5.ts
$ node parts1-5.js
   Determination would take 332.67857142857144 days to get to Mars.
   Determination would take 0.5683628571428571 days to get to the Moon.
```

Check your solution

28.7.5. Part 4 - Create a Spacecraft Class¶

Organize `getDaysToLocation` and the variables for name, speed, and miles per kilometer by moving them into a *class*.

1. Define a class named `Spacecraft`.
    1. Properties
        - `milesPerKilometer: number = 0.621;`
        - `name: string;`
        - `speedMph: number;`
    2. 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`.

    Note

    Once you complete the constructor, be sure to remove the variables you defined in part 1 (`spacecraftName`, `milesPerKilometer`, and `speedMph`.

2. Move the function `getDaysToLocation`, defined in Part 3, into the `Spacecraft` class.
    1. Remember to place the function after the constructor.
    2. Update the function to reference the class properties `this.milesPerKilometer` and `this.speedMph`.
3. Create an instance of the `Spacecraft` class.
    1. `let spaceShuttle = new Spacecraft('Determination', 17500);`
4. Print out the days to Mars.
    1. Use template literals, `${spaceShuttle.getDaysToLocation(kilometersToMars)}` and `${spaceShuttle.name}`.

Print out the days to the Moon.

5.
    1. Use template literals, `${spaceShuttle.getDaysToLocation(kilometersToTheMoon)}` and `${spaceShuttle.name}`.

6. Use the terminal in VSCode to recompile your `.ts` file, then run the `.js` file again.

Terminal

```
$ tsc parts1-5.ts
$ node parts1-5.js
   Determination would take 332.67857142857144 days to get to Mars.
   Determination would take 0.5683628571428571 days to get to the Moon.
```

Check your solution

28.7.6. Part 5 - Export and Import the SpaceLocation Class¶

1. From the file tree in VSCode, open the `SpaceLocation.ts` file.

2. Paste in the code provided below.

    1. Notice the `export` keyword. That is what allows us to import it later.

```
1 export class SpaceLocation {
2    kilometersAway: number;
3    name: string;
4
5    constructor(name: string, kilometersAway: number) {
6       this.name = name;
7       this.kilometersAway = kilometersAway;
8    }
9 }
```

3. Add the function `printDaysToLocation` to the `Spacecraft` class.

    1. Notice that it takes a parameter of type `SpaceLocation`.

```
1 printDaysToLocation(location: SpaceLocation) {
2    console.log(`${this.name} would take ${this.getDaysToLocation(loc
3 }
```

4. Import `SpaceLocation` into `parts1-5.ts`.

    1. Paste `import { SpaceLocation } from './SpaceLocation';` to the top of `parts1-5.ts`.

5. Replace the earlier `console.log` statements by using the class instance to print out the days to Mars and the Moon.

```
47 spaceShuttle.printDaysToLocation(new SpaceLocation('Mars', kilomete
48 spaceShuttle.printDaysToLocation(new SpaceLocation('the Moon', kilo
```

6. Use the terminal in VSCode to compile your `.ts` file, then run the `.js` file again.

Terminal

```
$ tsc parts1-5.ts
$ node parts1-5.js
   Determination would take 332.67857142857144 days to get to Mars.
   Determination would take 0.5683628571428571 days to get to the Moon.
```

Check your solution

28.7.7. Sanity Check¶

The `typescript-lc101-projects` repository has two branches---`master` and `solutions`. 'Nuff said.

- ← 28.6. Compiling TypeScript
- 28.8. Studio: TypeScript →

Back to top

LaunchCode logo

Search

1. Contents
2. 28. TypeScript
3. 28.8. Studio: TypeScript

28.8. Studio: TypeScript¶

Let's practice TypeScript by creating classes for rocket cargo calculations.

28.8.1. Starter Code¶

If you have not already done so, follow the instructions given in the TypeScript exercises to fork the GitHub repository.

Use the terminal to check that you are in the `master` branch, then navigate into the `studio` folder.

```
$ git branch
   * master
   solutions
$ pwd
   /typescript-lc101-projects
$ ls
   exercises       studio
$ cd studio
$ ls
   index.ts    Payload.ts
```

From the file tree in VSCode, open the `index.ts` file.

VSCode file tree for the TypeScript studio.

28.8.2. Requirements¶

1. Create classes for `Astronaut`, `Cargo`, and `Rocket`. (Details below).
    1. All classes should be defined in their own files.
2. Use the new classes to run a simulation in the `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 `massKg` property.

28.8.3. Classes¶

1. Create three new files---`Astronaut.ts`, `Cargo.ts`, and `Rocket.ts`. To do this in VSCode, click the "New File" button and enter the file name. Another option is to run the command `touch new_file_name` in the terminal.

   VSCode new file button.

2. Define each class (`Astronaut`, `Cargo`, `Rocket`) in a separate file. Each class should be exported using `export`.

   ```
   export class Astronaut {
      // properties and methods
   }
   ```

3. As needed, the classes can be imported using `import`.

   ```
   import { Astronaut } from './Astronaut';
   ```

28.8.3.1. Astronaut Class¶

1. Defined in `Astronaut.ts`
2. Implements the `Payload` interface
3. Properties
    1. `massKg` should be a number.

    2. `name` should be a string.
4. Constructor
    1. Parameter `massKg` should be a number.
    2. Parameter `name` should be string.
    3. Sets value of `this.massKg` and `this.name`.

28.8.3.2. Cargo Class¶

1. Defined in `Cargo.ts`
2. Implements the `Payload` interface
3. Properties
    1. `massKg` should be a number.
    2. `material` should be a string.
4. Constructor
    1. Parameter `massKg` should be a number.
    2. Parameter `material` should be a string.
    3. Sets value of `this.massKg` and `this.material`

28.8.3.3. Rocket Class¶

1. Defined in `Rocket.ts`.
2. Properties:
    1. `name` should be a string.
    2. `totalCapacityKg` should be a number.
    3. `cargoItems` should be an array of `Cargo` objects.
        - Should be initialized to an empty array `[]`
    4. `astronauts` should be an array of `Astronaut` objects.
        - Should be initialized to an empty array `[]`
3. Constructor:
    1. Parameter `name` should be a string.
    2. Parameter `totalCapacityKg` should be a number.
    3. Sets value of `this.name` and `this.totalCapacityKg`
4. Methods:
    1. `sumMass( items: Payload[] ): number`
        - Returns the sum of all `items` using each item's `massKg` property
    2. `currentMassKg(): number`
        - Uses `this.sumMass` to return the combined mass of `this.astronauts` and `this.cargoItems`
    3. `canAdd(item: Payload): boolean`
        - Returns `true` if `this.currentMassKg() + item.massKg <= this.totalCapacityKg`
    4. `addCargo(cargo: Cargo): boolean`
        - Uses `this.canAdd()` to see if another item can be added.
        - If `true`, adds `cargo` to `this.cargoItems` and returns `true`.
        - If `false`, returns `false`.
    5. `addAstronaut(astronaut: Astronaut): boolean`
        - Uses `this.canAdd()` to see if another astronaut can be added.
        - If `true`, adds `astronaut` to `this.astronauts` and returns `true`.

- If `false`, returns `false`.

28.8.4. Simulation in `index.ts`¶

Paste the code shown below into `index.ts`.

```
 1 import { Astronaut } from './Astronaut';
 2 import { Cargo } from './Cargo';
 3 import { Rocket } from './Rocket';
 4
 5 let falcon9: Rocket = new Rocket('Falcon 9', 7500);
 6
 7 let astronauts: Astronaut[] = [
 8    new Astronaut(75, 'Mae'),
 9    new Astronaut(81, 'Sally'),
10    new Astronaut(99, 'Charles')
11 ];
12
13 for (let i = 0; i < astronauts.length; i++) {
14    let astronaut = astronauts[i];
15    let status = '';
16    if (falcon9.addAstronaut(astronaut)) {
17       status = "On board";
18    } else {
19       status = "Not on board";
20    }
21    console.log(`${astronaut.name}: ${status}`);
22 }
23
24 let cargo: Cargo[] = [
25    new Cargo(3107.39, "Satellite"),
26    new Cargo(1000.39, "Space Probe"),
27    new Cargo(753, "Water"),
28    new Cargo(541, "Food"),
29    new Cargo(2107.39, "Tesla Roadster"),
30 ];
31
32 for (let i = 0; i < cargo.length; i++) {
33    let c = cargo[i];
34    let loaded = '';
35    if (falcon9.addCargo(c)) {
36       loaded = "Loaded"
37    } else {
38       loaded = "Not loaded"
39    }
40    console.log(`${c.material}: ${loaded}`);
41 }
42
43 console.log(`Final cargo and astronaut mass: ${falcon9.currentMassKg()}
```

28.8.5. Compile and Run `index.ts`¶

1. Use the terminal in VSCode to compile your `index.ts` file. This will also compile the modules you imported into the file (`Astronaut.ts`, `Rocket.ts`, etc.).
2. Use the command `node index.js` to run the JavaScript file created during the build process.

```
$ ls
   Astronaut.ts    Cargo.ts        Payload.ts      Rocket.ts       index.t
$ tsc index.ts
$ ls
   Astronaut.js    Cargo.js        Payload.js      Rocket.js       index.j
   Astronaut.ts    Cargo.ts        Payload.ts      Rocket.ts       index.t
$ node index.js
```

28.8.5.1. Expected Console Output¶

```
Mae: On board
Sally: On board
Charles: On board
Satellite: Loaded
Space Probe: Loaded
Water: Loaded
Food: Loaded
Tesla Roadster: Not loaded
Final cargo and astronaut mass: 5656.78 kg.
```

28.8.6. Submitting Your Work¶

1. Once you have your project working, use the terminal to commit and push your changes up to your forked GitHub repository.
2. Login to your account and navigate to your project. Copy the URL.
3. In Canvas, open the TypeScript studio assignment and click the "Submit" button. An input box will appear.
4. Paste the URL for your GitHub project into the box, then click "Submit" again.

- ← 28.7. Exercises: TypeScript
- 29. Angular, Part 1 →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1

29. Angular, Part 1¶

- ← 28.8. Studio: TypeScript
- 29.1. Why Use JavaScript Libraries →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.1. Why Use JavaScript Libraries

29.1. 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.

- ← 29. Angular, Part 1
- 29.2. Templates →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.2. Templates

29.2. 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.

29.2.1. 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>`, `<img>`, 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...

29.2.2. 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.

29.2.2.1. No Template¶

To demonstrate how helpful templates can be, you will need a copy of the LaunchCode Angular projects repository:

1. Fork the angular-lc101-projects repository on GitHub.
2. Clone your fork. If you need a reminder for how to do this, refer to the Git studio.

The repository contains a lot of angular code that we will soon go over. For now, navigate to *lesson1/examples/noTemplate*.

```
$ pwd
   angular-lc101-projects
$ ls
   lesson1 lesson2 lesson3
$ cd lesson1
$ ls
   exercises examples
$ cd examples/noTemplate
```

The code in `noTemplate/index.html` builds a simple 3-column web page. It defines the location for each heading, unordered list, and button. The `noTemplate/style.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<
36       <a href="https://www.webelements.com/" target="_blank">WebElement
37    </div>
38 </body>
```

Copy the file path for `noTemplate/index.html` and enter the address into your browser. You'll see a page like this:

A uniquely styled web page.

A uniquely styled web page.¶

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.

29.2.2.2. 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.

```
<body>
   <h1>{{mainHeading}}</h1>
   <div class="row list">
      <div class='movie'>
         <h4>Movies</h4>
         <ul>{{movieTitles}}</ul>
         <button class='movie'>More</button>
      </div>
      <div class='school'>
         <h4>Education</h4>
         <ul>{{schoolNames}}</ul>
         <button class='school'>More</button>
      </div>
      <div class='hobby'>
         <h4>Hobbies</h4>
         <ul>{{hobbies}}</ul>
         <button class='hobby'>More</button>
      </div>
   </div>
   <hr>
   <div class="links">{{headingAndLinkList}}</div>
</body>
```

Tip

Don't change your HTML in `noTemplate/index.html`. We haven't yet covered the Angular work needed to make use of this templating syntax.

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 `<li></li>` tags.

By defining our template in an even more general manner, we could replace the `h4`, `ul` and `button` structure with a single placeholder.

```
<body>
   <h1>{{mainHeading}}</h1>
   <div class="row list">
      <div class='movie'>{{movieContent}}</div>
```

```
 5       <div class='school'>{{schoolContent}}</div>
 6       <div class='hobby'>{{hobbyContent}}</div>
 7    </div>
 8    <hr>
 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.

29.2.2.3. Templates Support Dynamic Content¶

If we add a search box to our website, a user could enter *NASA images*, *giraffe gif*, *move 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 web page that responds to a changing set of data, especially if that data is unknown to us.

29.2.3. Templates Provide Structure, Not Content¶

Templates allow us to decide where to display data on our web page, even if we do not know exactly what that data will be. Information pulled from forms, APIs, or user input will be formatted to fit within our design.

Visual of a template structure.

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.

29.2.4. Check Your Understanding¶

Question

Why should we use a template to design a web page 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

- ← 29.1. Why Use JavaScript Libraries
- 29.3. Angular File Structure →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.3. Angular File Structure

29.3. Angular File Structure¶

Each Angular project contains a standard file structure, which is shown in the figure below.

Visual of the standard Angular file structure.

All of the project files (and there will be LOTS) are stored in the single, top-level folder, `project-name`, 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 five important files: `index.html`, `main.ts`, `style.css`, `tests.ts` 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`.

29.3.1. Installing Angular¶

Angular uses its own set of command line instructions to create, update, and launch projects. Before you dive too deeply into the Angular lessons, you need to install the Angular command line interface (CLI) on your computer. Fortunately, the process is relatively painless.

Open up the terminal (or the terminal panel in VSCode) and enter the following command:

```
npm install -g @angular/[email protected]
```

Note

There are more recent versions of this package but we're using version 12 for generating new code. When you pull down starter code from Github, you will have to run `npm install` which will set up that project to use Angular 8. While you may notice differences between the two versions, the core functionality of Angular will remain the same.

Note

If the installer prompts you to make choices, just accept all of the default options.

This command installs the CLI *globally* on your computer, which means that Angular commands will work regardless of the folder you have open.

Angular commands begin with the keyword `ng` (for A-ng-ular), and the most commonly used include:

1. `ng new`: Creates a new Angular project in the current directory. The shortcut syntax is `ng n`.
2. `ng generate`: Creates new files within an existing project. The shortcut syntax is `ng g`.
3. `ng serve`: Compiles a project and *launches* it in a form that can be displayed in a browser. The shortcut syntax is `ng s`.

For a complete list of commands, refer to the Angular documentation.

29.3.1.1. Check Your GitHub Account¶

For the "HTML Me Something" assignment, you created an account on GitHub . Since you will modify Angular projects over several lessons, you need access to GitHub to download starter code and store your progress.

Follow the link and make sure you remember your login information, or enroll if you have not yet created an account.

29.3.1.2. What's in a Name?¶

If you Google "Angular tutorial", you will receive plenty of hits. Some of the resources will be exceptional, others not. However, you will probably notice that many of the results refer to *AngularJS*. This is a previous version of the software, and it is NOT the same as modern Angular.

AngularJS is NOT the same as Angular. Even the websites are different (angularjs.org vs. angular.io). We recommend that you avoid AngularJS resources for now. You can always learn how to use the old version later if your job requires it.

29.3.2. Ready To Go¶

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.

- ← 29.2. Templates
- 29.4. Angular First Steps →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.4. Angular First Steps

29.4. Angular First Steps¶

The goal here is to create a minimum working template for a web page. It will serve as a jumping-off point for your later exercises and studios, so take as much time as you need to get comfortable with the basics.

29.4.1. Starting a New Project¶

First, use the terminal to create a folder for all of your Angular projects:

```
$ mkdir angular_practice
```

Each new project you create will have its own folder inside this container.

Note

You can use either the standalone Terminal application on your computer or the built-in Terminal panel in VSCode.

For Angular projects, some programmers find it more convenient to work entirely inside VSCode or another code editor. The IDEs provide a visual tree for navigating between files and folders, and Angular generates LOTS of these.

Next, navigate into the new folder and create your first Angular project.

```
$ cd angular_practice
$ ng new my-project-name
```

This process will take some time, so be patient. You will also be prompted twice:

Visual of the prompts for creating a new Angular project.

Answer "No" to the routing question, and select the default "CSS" for the stylesheet by pressing "Enter".

29.4.1.1. Examine the Files Created¶

File tree for a new Angular project.

1. The `src` folder holds the files and source code needed for the project.
2. The `app` folder holds the content for the web page. 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.

29.4.1.2. What To Ignore¶

For every new project, Angular automatically sets up the code to make the different parts communicate with each other. As your skills grow during your career, you may need to learn how to modify these files. For now, however, leave the following alone:

1. `main.ts`, `test.ts`, and `polyfills.ts`. No touch!
2. The `e2e`, `node_modules`, and `environments` folders. No touch!
3. `.json` files. No touch!

The `assets` folder is a "Maybe touch". It holds user defined files that support a project. Examples include JavaScript code, images, gifs, or video clips. The exercises in the next chapter will use files stored in this folder, but we will leave it empty for this lesson.

29.4.2. Launch the Page¶

`ng new` creates all of the files required to launch a functioning web page. You have not added any content yet, but Angular provides a standard starting point that allows you to check if everything works.

Before sending an Angular project out into the world, you should preview it *locally* in a browser. Any changes you make to the project files will be reflected only on your screen, so you can play around with the code without worry.

Note

Even though you view your work in a browser, "local" means no one else can access your webpage, since it is stored on your computer. If you want to share your local page with someone else, they will have to look over your shoulder.

To *launch* your new webpage, use the Terminal to navigate into the project folder, then enter the command `ng serve`.

```
$ cd my-project-name
$ ng serve
```

Be prepared to wait... The compiler requires some time to build and deploy even small projects, and it may be a few moments before you see any action in the terminal. If no errors occur, a "Compiled successfully" message eventually appears.

The important part of this feedback is the `localhost` line, which provides a URL for viewing your work in a browser. Copy the URL and paste it into the address bar of your web browser (Mac shortcut: command-click automatically opens the URL in your default browser).

Terminal feedback after ``ng serve``.

29.4.2.1. What is `ng serve` Doing?¶

`ng serve` allows you to view your project in a browser by running a series of tasks in the background. It's NOT magic, just the tedious mechanics that you don't need to set up yourself.

`ng serve` performs these tasks:

1. Compiles and analyzes your Angular files to *build* HTML and JavaScript files that can be run in a browser.
    1. This step will throw errors if you try to serve code that contains syntax or other errors.
    2. You will learn more about the different types of files that are compiled in the coming sections.
2. Starts a web server on your computer that serves the built version of your Angular project.
    1. Your Angular project is viewable at the web address http://localhost:4200

Note

Angular projects are written in TypeScript. Your web browser can run HTML, CSS, and JavaScript. In order for your Angular project to run in the browser, the TypeScript code has to be converted into JavaScript. The conversion from TypeScript to JavaScript happens during the build phase of `ng serve`.

29.4.3. Yay! A Webpage!¶

Congratulations! You have a functioning webpage. You should see the following in your browser:

Angular new app default page.

This is the default format created by `ng new`, but your chosen project name will replace `my-project-name` in the title. The links lead back to selected pages from the angular.io documentation.

Feel free to play around a little bit before continuing. Do not worry about breaking anything. If necessary, you can always start another new project.

29.4.3.1. Try It¶

In VSCode, open the four files within the `app` folder. Modify the code to accomplish the following:

1. Find where your project name is assigned to the `title` variable. Replace it with a different string.
2. Change one `h2` heading to an `h3`.
3. Change the color for the *Welcome to...* heading.
4. Change one of the links to send users to your favorite website.
5. Replace the Angular shield with a different image.

After making each change, save your work. Your webpage should automatically refresh.

Note

Which files did you modify?

Do not worry if you got stuck on some of the tasks. This was a time for experimentation. As long as you tried something and saw the result, you still learned something valuable.

29.4.4. Un-launching Your Page¶

`ng serve` continues to run until you press *control+c* in the terminal. Go ahead and interrupt the process now. If you try refreshing your page, you will see an error.

Now let's take a look at the different project files.

- ← 29.3. Angular File Structure
- 29.5. The Angular Framework →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.5. The Angular Framework

29.5. The Angular Framework¶

In VSCode, navigate to the `src` folder, open the `index.html` file, and examine the code:

```
 1 <!doctype html>
 2 <html lang="en">
 3 <head>
 4    <meta charset="utf-8">
 5    <title>FirstProject</title>
 6    <base href="/">
 7
 8    <meta name="viewport" content="width=device-width, initial-scale=1">
 9    <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12    <app-root></app-root>
13 </body>
14 </html>
```

You have seen most of the HTML tags before, but line 12 shows something odd. The strange tag `<app-root>` represents a key idea behind building

templates. Angular allows us to define our own tags, which are used as another type of placeholder in an HTML file. In this case, `<app-root></app-root>` reserves space on the web page for information supplied by other files. Line 12 essentially says, "Display all the content from the `app` folder here."

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 web page 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.

Most of our work with Angular will take place within the `app` folder, so let's take a closer look at some of the files there.

29.5.1. Inside the app folder¶

One way to change the color of the *Welcome to...* heading would be to open the `app.component.css` file and add some styling:

```
1 h1 {
2    color: brown;
3 }
```

We can freely modify this file, but the CSS instructions only affect the HTML files within `app`. Also, the code in `app.component.css` overrides any CSS found in the higher level `styles.css` file.

This is the 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 three other `app` files.

29.5.1.1. `app.component.html` File¶

Example

Here is a sample of the default code inside `app.component.html`:

```
 1 <div style="text-align:center">
 2    <h1>
 3       Welcome to {{ title }}!
 4    </h1>
 5    <img width="300" alt="Angular Logo" src="image path...">
```

```
 6 </div>
 7 <h2>Here are some links to help you start: </h2>
 8 <ul>
 9   <!-- List items here... -->
10 </ul>
```

`app.component.html` contains the structure and most of the text seen on the "Welcome to..." page. Note the placeholder `{{title}}` in line 3. This gets filled with data passed in from another file, and it allows us to modify the content on the page without revising the HTML.

`app.component.html` serves as the main template for your web page. This file will usually NOT hold a lot of HTML code. Instead, it will contain many placeholders for content defined elsewhere in the project.

Later in this chapter, you will learn how to add new components to the `app` folder as well as how to arrange them in the HTML file.

29.5.1.2. `app.component.ts` File¶

Example

`app.component.ts`

```
 1 import { Component } from '@angular/core';
 2
 3 @Component({
 4   selector: 'app-root',
 5   templateUrl: './app.component.html',
 6   styleUrls: ['./app.component.css']
 7 })
 8 export class AppComponent {
 9   title = 'my-project-name';
10 }
```

`app.component.ts` performs several important functions with very few lines.

1. Line 4 defines the tag `<app-root>`, which we saw in line 12 of `index.html`. The tag can also be used in any files that import the `AppComponent` class.
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. (If you set a different color for the *Welcome to...* sentence in the Try It tasks, this is why changing the css file worked).
4. Line 8 makes the `AppComponent` class available to other files.

Take a look at `app.component.html` again. We mentioned the `{{title}}` 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 string `'my-`

project-name' to the title variable. Changing 'my-project-name' to a different value alters the web page.

29.5.1.3. app.module.ts File¶

Example

app.module.ts

```
 1 import { BrowserModule } from '@angular/platform-browser';
 2 import { NgModule } from '@angular/core';
 3
 4 import { AppComponent } from './app.component';
 5
 6 @NgModule({
 7   declarations: [ AppComponent ],
 8   imports: [ BrowserModule ],
 9   providers: [],
10   bootstrap: [AppComponent]
11 })
12 export class AppModule { }
```

Just like before, there is a lot going on within very few lines.

1. Lines 1, 2, and 8 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. Line 4 imports the class AppComponent from the local file app.component.ts.
3. Line 4 also pulls in references to any other files linked to app.component.ts.
4. Line 7 declares the imported local files as necessary for the project.
5. Line 12 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.

29.5.2. Change The Content¶

Enough detail. Let's explore some more.

If you did not complete all of the Try It tasks on the previous page, attempt them now. After that...

29.5.2.1. Try It!¶

1. Run `ng serve` in the terminal to launch your web page again.
2. In `app.component.ts`, declare and assign two variables in the `AppComponent` class---`name` and `itemList`.
    1. `name` holds your name.
    2. `itemList` is an array holding at least 4 items.

```
1 export class AppComponent {
2    name: string = 'Barbara Liskov';
3    itemList: string[] = ['item1', 'item2', 'item3', 'item4'];
4 }
```

Note

Instead of using the strong TypeScript variable declarations in step 2, we could substitute a pattern more like JavaScript:

```
1 export class AppComponent {
2    name = 'Brendan Eich';
3    itemList = ['item1', 'item2', 'item3', 'item4'];
4 }
```

3. Replace line 4 in `app.component.html` with `<h1>{{name}}'s First Angular Project</h1>`. Save your work and then check to make sure the web page shows the new heading.
4. Modify the `<li></li>` elements in `app.component.html` to display the elements from `itemList` in an unordered list. Be sure to use placeholders like `{{itemList[0]}}` between the tags.
5. Return to the `AppComponent` class in the `.ts` file. Define a `rectangle` object 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.

```
1 rectangle = {
2    length: 5,
3    width: 6,
4    area: function() {
5       return this.length * this.width;
6    }
7 }
```

6. Add a `<p></p>` element in `app.component.html` to display the sentence, "The rectangle has a length of ___ cm, a width of ___ cm, and an area of

___ cm^2." Replace the blanks with placeholders so the web page displays the correct numbers whenever `length` or `width` are changed.

```
<p>The rectangle has a length of {{rectangle.length}} cm, a width of {
   and an area of {{rectangle.area()}} cm^2.</p>
```

29.5.3. Filename Pattern¶

Each of the files in the `app` folder contain the word `component` in their name. This results from the fundamental idea behind Angular. Each *template* for a web page 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.

29.5.4. Check Your Understanding¶

Question

Where would be the BEST place to modify our code if we want a different font for any <p> text within a 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 our 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`

- ← 29.4. Angular First Steps
- 29.6. Components →

Back to top

29.6. Components¶

In Angular a **component** controls one part of the page, called a *view*.

Angular builds a web page by combining multiple components together. Splitting our page into individual components makes our application more organized. It also 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.

29.6.1. Start Fresh¶

In the terminal, navigate back to the `angular_practice` folder you created on the Angular First Steps page. Create a new project called `component-practice`.

```
$ ng new component-practice
```

Open the `app.component.html` file in VSCode. Remove ALL of the code and replace it with an empty `div` element.

```
1 <div>
2
3 </div>
```

OK, the stage is set for a closer look at components.

29.6.2. Component Files¶

Angular components consist of 4 files:

1. an HTML file (.html)
2. a CSS file (.css)
3. a typescript file (.ts)
4. a test file (.spec.ts)

Visual of the files associated with a component.

Visual of file tree after adding a new component, `header`, to the project.¶

Looking at the file tree, we see that all four files contain the name of the component---`header`, in this case. Also, the files are located in a folder named after the component.

If we add a new component named `task-list`, the four files created inside the `task-list` folder would be called:

1. `task-list.component.html`
2. `task-list.component.css`
3. `task-list.component.ts`
4. `task-list.component.spec.ts`

Each file contains information specific to that component. `task-list.component.html` holds the HTML required for the task-list and no other component. `task-list.component.css` only styles html within the `task-list` folder, the typescript code in `task-list.component.ts` only applies to this component, and all the tests for `task-list` will be found in `task-list.component.spec.ts`.

29.6.3. Adding a New Component¶

Each component is a smaller part of an overall web application. The main component, `app`, serves as a base structure, and it comes standard with all Angular applications. It is the container that holds all of the other components, and it organizes them into the web application.

When you generate a new component using the Angular CLI, it is automatically added to `app`. Let's explore how this process works.

29.6.3.1. `ng generate`¶

To create a new Angular component, the terminal syntax is:

```
$ ng generate component component-name
```

Warning

A common mistake is to create a new component in the wrong location, say in the `src` folder instead of in `app`.

`ng generate` places the new component folder within your current directory. Use the terminal to navigate to where you want the component to go BEFORE running the generate command.

Creating a `task-list` component looks something like this:

Visual of the terminal command to create a new Component.

Terminal output when creating a new component.¶

From the output, we see that the `ng generate` command created four new files in the `src/app/task-list` folder.

Note

Recall that `ng generate` can be shortened to `ng g`.

29.6.3.2. Try It¶

1. Use the terminal panel in VSCode to navigate into the `app` folder.

```
$ ls
   first-project  component-practice
$ cd component-practice
$ cd src
$ cd app
$ ls
   app.component.css    app.component.spec.ts   app.module.ts
   app.component.html   app.component.ts
```

2. Run `ng generate component task-list`.

3. Add a `header` component by running `ng generate component header`.

When done, your file structure in VSCode should look something like:

Visual of the results of generate Component.

File tree of the newly added `task-list` component.¶

29.6.4. `app.module.ts`¶

In order to communicate with the new components, `app.module.ts` needs new `import` statements. Fortunately, `ng generate` updates the code automatically. We do not need to worry about taking care of this task ourselves.

Before `ng generate`:

```
 1 import { BrowserModule } from '@angular/platform-browser';
 2 import { NgModule } from '@angular/core';
 3
 4 import { AppComponent } from './app.component';
 5
 6 @NgModule({
 7   declarations: [ AppComponent ],
 8   imports: [ BrowserModule ],
 9   providers: [],
10   bootstrap: [AppComponent]
```

```
11 })
12 export class AppModule { }
```

After generating the `header` and `task-list` components:

```
 1 import { BrowserModule } from '@angular/platform-browser';
 2 import { NgModule } from '@angular/core';
 3
 4 import { AppComponent } from './app.component';
 5 import { TaskListComponent } from './task-list/task-list.component';
 6 import { HeaderComponent } from './header/header.component';
 7
 8 @NgModule({
 9    declarations: [
10       AppComponent,
11       TaskListComponent,
12       HeaderComponent
13    ],
14    imports: [ BrowserModule ],
15    providers: [],
16    bootstrap: [AppComponent]
17 })
18 export class AppModule { }
```

Angular updates `app.module.ts` by adding new `import` statements on lines 5 and 6 as well as expanding the `declarations` array on line 9.

Note

Generating new components automatically updates `app.module.ts`. However, if you *delete* a component, you must MANUALLY remove its `import` statement and its name in the `declarations` array.

29.6.5. Arranging Components¶

Run `ng serve` to launch the webpage. The page shows up empty because we removed all of the code from `app.component.html` except for the `div` tags.

Modify `app.component.html` as follows:

```
1 <div>
2    <app-header></app-header>
3 </div>
```

Save your change and wait for the webpage to refresh. You should now see the text "header works!" at the top of the page.

Confirmation that the header component works.

This is another helpful feature with Angular---when you correctly implement a new component, confirmation text appears on the screen.

How did <app-header></app-header> make this happen? Open header.component.ts in VSCode:

```
 1 import { Component, OnInit } from '@angular/core';
 2
 3 @Component({
 4    selector: 'app-header',
 5    templateUrl: './header.component.html',
 6    styleUrls: ['./header.component.css']
 7 })
 8 export class HeaderComponent implements OnInit {
 9
10    constructor() { }
11
12    ngOnInit() {
13    }
14 }
```

Line 4 defines the HTML tag for the header component to be app-header. If we try changing the string to 'orange', we would see the "header works!" text disappear from the webpage. This is because the HTML tag <app-header> is no longer linked to the component. The string assigned in line 4 MUST match the tags used in app.component.html.

29.6.5.1. Modify the Header Text¶

Open header.component.html in VSCode:

```
1 <p>header works!</p>
```

Ah ha! The HTML file contains the text that appeared on our webpage. Anything added to this file will appear between the <app-header></app-header> tags in app.component.html.

Try It

1. Replace line 1 in header.component.html with:

```
1 <h1>My header works!</h1>
2 <p>This is not a header, but I'm adding it anyway.</p>
3 <div style="text-align: center">
4    <h2>Look! A centered h2.</h2>
5    <p>More centered text.</p>
```

```
6 </div>
7 <p>Not centered text.</p>
```

Save your code and refresh the page. How does its appearance change?

2. What happens if we use TWO `<app-header>` elements in `app.component.html`? Try it to find out.

```
1 <div>
2    <app-header></app-header>
3    <app-header></app-header>
4 </div>
```

29.6.5.2. Bring in `task-list`¶

Line 4 in `header.component.ts` defined the `app-header` tag, and line 4 in `task-list.component.ts` does something similar.

Modify `app.component.html` as follows:

```
1 <div>
2    <app-header></app-header>
3    <app-task-list></app-task-list>
4 </div>
```

Your webpage should look similar to:

Two components added to the app template.

Confirmation that `task-list` is properly working in your project.¶

Try It

Move `<app-task-list></app-task-list>` above `<app-header></app-header>` and see how the webpage changes.

29.6.5.3. THIS IS WHY TEMPLATES ARE AWESOME!¶

Trying to correctly format and place content on a webpage can be difficult, especially if you need to present lots of data or mix different formatting styles for headings, lists, plain text, etc.

Rather than deal with our header, task-list, and other content at the same time, creating components allows us to:

1. Create a simple HTML file that serves as a framework.
2. Format each piece of our content separately, without worrying about how that formatting affects other parts of the webpage.

3. Easily add content to the framework by using custom HTML tags.
4. Quickly relocate the components on a page just by rearranging their custom tags.

29.6.6. Component Nesting¶

Components can be put inside of other components. In essence, this is how the `app` component works. It is the component that holds all other components.

However, sometimes you might want to nest a new component inside of another one rather than in `app`.

Let's assume we want to add a new component within our `task-list` folder. In this case, we navigate into the `task-list` directory and then run the `ng generate component` command.

```
$ ls
   app.component.css    app.component.spec.ts   app.module.ts   task-list
   app.component.html   app.component.ts        header
$ cd task-list
$ ng generate component inside-task-list
```

Running this command nests our new folder inside of the `task-list` folder, and it contains the four files we would expect.

Visual of the result of the running the commands to create a nested component.

Nested components.¶

When we place one component inside of another, we must pay attention to how the components interact. The nested component is called the *child*, while the original component is called the *parent*. In our example, `task-list` serves as the parent, while `inside-task-list` is the child.

1. Any CSS, HTML, or JavaScript we write for the nested component (the child) only affects that component. Changes to the child do NOT affect the parent.
2. The parent component DOES influence the nested one. For example, any CSS within `task-list.component.css` applies to both `task-list.component.html` AND `inside-task-list.component.html`.
3. If we want `inside-task-list` to have different styling, we need to add code to `inside-task-list.component.css` to override the parent.

29.6.7. Check Your Understanding¶

If you have not already done so, use `ng generate` to *nest* the `inside-task-list` component inside the `task-list component`.

Question

EXPERIMENT! Discover.

Where could we place the `<app-inside-task-list></app-inside-task-list>` element to make "inside-task-list works!" appear on the screen? Select ALL options that work.

1. Place the element in `app.component.html`.
2. Place the element in `task-list.component.html`.
3. Place the element in `inside-task-list.component.html`.
4. Place the element in `index.html`.

- ← 29.5. The Angular Framework
- 29.7. Exercises: Angular, Lesson 1 →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.7. Exercises: Angular, Lesson 1

29.7. Exercises: Angular, Lesson 1¶

The following exercises walk you through the process of modifying existing components and generating new ones.

29.7.1. Starter Code¶

For this set of exercises, you will need a copy of the LaunchCode Angular projects repository. If you haven't done so already,

1. Fork the angular-lc101-projects repository on GitHub.
2. Clone your fork. If you need a reminder for how to do this, refer to the Git studio.

Navigate into the lesson 1 exercises now:

```
$ pwd
   angular-lc101-projects
$ ls
   lesson1 lesson2 lesson3
$ cd lesson1
$ ls
   exercises examples
$ cd exercises
```

The repository contains code but NOT the Angular modules necessary to launch the webpage. In the terminal, run `npm install` to add these modules. Next, enter `ng serve` and open the `localhost` URL in your browser.

The starter page should look like this:

Starting setup for exercises.

Starting set up for the exercises. Lots of content is missing.¶

The page is not pretty yet, but you get to fix that.

29.7.2. 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 for the heading and list items can be any color EXCEPT black. (HINT: Take advantage of the `movies` class).
    2. The movie list should have a centered heading.
    3. The font size should be large enough to easily read.

    Check your solution

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, but not black.
    3. The chores list should have an underlined heading.
    4. The chores in the list should be italicized.

29.7.2.1. Add More Movies¶

Browser screen shot showing list of movies that contains "The Manchurian Candidate" and "Oceans 8".

The list of movies is built using an array defined in `movie-list.component.ts`:

```
export class MovieListComponent implements OnInit {
   movies = ['The Manchurian Candidate', 'Oceans 8'];
```

The titles in the `movies` array are referenced in the template `movie-list.component.html` by adding *placeholders* in the HTML.

You can put almost any valid JavaScript inside the `{{ }}` in an Angular template. For example, `{{ movies[0] }}` references the `movies` array, and `movies[0]` returns the first item in the array.

```
<div class="movies">
   <h3>Movies to Watch</h3>
   <ol>
      <li>{{ movies[0] }}</li>
      <li>{{ movies[1] }}</li>
   </ol>
</div>
```

Using references in your HTML allows you to easily modify the content on your webpage. Instead of typing specific movie titles between the tags every time the list changes, references allow us to modify the data in one easy to find array.

1. Add two more items to the `movies` array.

   Check your solution

2. Add two more `<li></li>` elements to `movie-list.component.html` and use placeholders to reference the new movies in the array.

29.7.2.2. Complete the `fav-photos` Component¶

1. 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 the `photosTitle` variable.

   2. The `image` variables should hold URLs for images, but only one is is filled in. Complete at least one more, which can be from the web or personal pictures. To copy the URL for an image on the web, right-click (or control-click) on the image and select the "Copy Image Location" menu option.

   3. In the `.html` file for this component, use placeholders in the `img` tags to display your chosen images.

      ```
      <img src="{{placeholder}}" alt="Oops! Missing photo!">
      ```

   4. Adjust the HTML to display one image per line.

   5. Use the `.css` file for this component to make all the images be the same size.

   6. Refresh the webpage to check the updated content.

   Check your solution

Before moving on, save and commit your work.

29.7.3. 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.

1. The page needs a title.
    1. Use the terminal to navigate into `app` folder.
    2. Generate the new component `page-title`:

        ```
        ng generate component page-title
        ```

    3. Open `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`.
    4. In the `PageTitleComponent` class, define a `title` variable and assign it a string.
    5. 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.
    6. Add the `<page-title></page-title>` element to `app.component.html`.
    7. Save all of your changes and refresh the page to see your new content.
2. The page needs a set of links to favorite websites.
    1. Generate a `fav-links` component. Open `fav-links.component.ts` and shorten the tag name to just `fav-links`.
    2. In the `FavLinksComponent` class, define the variable `favLinks` and assign it an array that contains two or more URLs.
    3. In the `.html` file for this component, add a set of `<a>` tags for the web links. Each link should be on its own line.
    4. Inside each `<a>` tag, set the `href` attribute equal to a placeholder for an element in the `favLinks` array:

        ```
        <a href = "{{placeholder}}">Link text...</a>
        ```

    5. Add `<fav-links></fav-links>` to `app.component.html`. Save all of your changes, then refresh the page to see your new content.

Check your solution

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.

Angular automatically takes care of updating `app.module.ts` when you generate new components. However, *deleting* a component does NOT remove the references from the file.

29.7.4. Part 3: Rearrange the Components¶

The content on the page appears quite jumbled, since we gave you no guidance on where to put the custom tags in `app.component.html`. Fortunately, templates allow us to easily move items around the framework.

1. 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.

    Check your solution

Your final page should have this format (the dashed lines are optional):

Angular Lesson 1 Exercises project.

29.7.4.1. Optional Final Touches¶

1. 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.

29.7.5. Sanity Check¶

The `angular-lc101-projects` repository contains two branches:

1. A `master` branch with all the starter code for lessons 1, 2, and 3.
2. A `solutions` branch with completed code.

If you get stuck on a particular exercise:

1. Try again.
2. Try again again.
3. Ask your TA, instructor, classmates, or Google for tips.

4. Try again.
5. Take a break and give your brain a chance to rest.
6. Try again.
7. Feel completely justified in switching to the `solutions` branch to check the code.

Note

If you jumped right to step 7, you missed out on a stellar learning opportunity.

Angular Lesson 1 results.

Customized and complete angular project example.¶

- ← 29.6. Components
- 29.8. Studio: Angular, Part 1 →

Back to top

LaunchCode logo

Search

1. Contents
2. 29. Angular, Part 1
3. 29.8. Studio: Angular, Part 1

29.8. Studio: Angular, Part 1¶

In this chapter, you learned about the Angular file structure, templates, and components. Over the next three classes, you will build a Mission Planning Dashboard using your Angular skills.

29.8.1. Mission Planning Dashboard¶

A useful and common front end application is a *dashboard*. It shows a summary of information about a topic, helping users of the web app make informed decisions.

You will create a *Space Mission Planning Dashboard*.

29.8.2. Create Angular Project¶

1. Launch Visual Studio Code. If you created an `angular_practice` folder earlier in the chapter, use the *File* menu to open it. If you did not create a practice folder, make one!

2. Open a terminal at the root of your `angular_practice` folder.

3. Create a new Angular project by running `ng new angular-studio-part1`.

    1. When prompted about using routing, enter "N" for No.
    2. When prompted to select the stylesheet format, select CSS.

    ```
    $ ng new angular-studio-part1
    ? Would you like to add Angular routing? No
    ? Which stylesheet format would you like to use? CSS
    ```

4. When the process finishes, use the *File* menu to open `angular-studio-part1` in VS Code.

5. In the terminal, use `pwd` to check your position in the file tree. If necessary, navigate into the `angular-studio-part1` directory.

6. Install dependencies by running `npm install`

7. Verify that the application will run by running `ng serve`

8. View the site in your browser at http://localhost:4200

    1. You should see a header that says "Welcome to angular-studio-part1!"

9. In the terminal, initialize your project with git, then stage and commit your files locally before starting on the features.

Tip

You will likely still have `ng serve` running in your terminal. You can stop `ng serve` by pressing the keys "**control+c**", or you can open an additional terminal window to run the git commands.

29.8.3. Requirements¶

The mission dashboard you are creating will eventually look like this.

Screen shot showing the mission dashboard with mission name, rocket name, crew members, equipment, and experiments.

29.8.3.1. Update Starter Page Content¶

The default stater page created by Angular contains default text, images, and links. Your job is to remove the default content.

1. Clear out the contents of file `app.component.html`.
2. Type in the text `Add components here...` into `app.component.html`.
3. Run `ng serve` if it's not already running.
4. View the app in your browser to verify the words "Add components here..." is the only thing that appears on the page.

29.8.3.2. Header Component¶

You need to create a component that shows the title, mission name, and carrier rocket.

1. In terminal navigate to the folder `src/app` folder.
2. Create a header component using `ng g component header`.
3. In the file `header.component.html` add HTML:

```
<h1>Mission Planning Dashboard</h1>
<h3>Mission Name: {{ missionName }}</h3>
<h3>Carrier Rocket: {{ rocketName }}</h3>
```

1. Add the variables `missionName` and `rocketName` to the header component in `header.component.ts`

```
export class HeaderComponent implements OnInit {

    missionName: string = "Mars 2030";
    rocketName: string = "Plasma Max";
```

1. Add a reference to the header component in `app.component.html`.

```
<app-header></app-header>
```

1. View the app in your browser to verify that the title, mission name, and rocket name are visible.

Screen shot of browser showing address localhost:4200, which has a title of Mission Planning Dashboard, a Mission Name, and a Carrier Rocket.

29.8.3.3. Crew Component¶

Next you need to make a component to show a list of crew members.

1. Create the component by running `ng g component crew`.

2. Set the contents of `crew.component.html` to be:

```
<h3>Crew</h3>
<ul>
   <li>Jessica Watkins</li>
   <li>Raja Chari</li>
   <li>Jasmin Moghbeli</li>
</ul>
```

1. Add a reference to the header component in `app.component.html`.

```
<app-header></app-header>
<app-crew></app-crew>
```

Screen shot of browser showing address localhost:4200, which has a title of Mission Planning Dashboard, a Mission Name,a Carrier Rocket, a Crew header, and a list of crew members in an unordered list.

29.8.3.4. Equipment Component¶

Now you need to create a component to show a list of equipment.

1. Create an equipment component named `equipment`.
2. The component should display the following:
    1. An `<h3>` that contains "Equipment"
    2. A `<ul>` that contains `<li>` for: Habitat dome, Drones, Food containers, Oxygen tanks
3. Add the equipment component to `app.component.html` using the HTML below. Notice the `<div>` surrounding the crew and equipment components.

```
<app-header></app-header>
<div class="box">
   <app-crew></app-crew>
   <app-equipment></app-equipment>
</div>
```

1. Add CSS to file `app.component.css` to horizontally align the crew and equipment lists. Without this CSS, the equipment list will appear below the crew list.

```
.box {
   display: flex;
   padding: 10px;
}
```

Screen shot of browser showing address localhost:4200, which has a title of Mission Planning Dashboard, a Mission Name, a Carrier Rocket, a Crew header, a list of crew members, and a list of equipment.

Note

A full explanation of `display: flex;` is beyond the scope of this book. For more information see MDN flex box docs and CSS Tricks flex box guide.

29.8.3.5. Experiments Component¶

1. Finally, add an experiments component that contains the HTML below:

```
<h3>Experiments</h3>
<ul>
   <li>Mars soil sample</li>
   <li>Plant growth in habitat</li>
   <li>Human bone density</li>
</ul>
```

2. Make the list of experiments show up to the right of equipment list.

When done your dashboard should look like this:

Screen shot showing the mission dashboard with mission name, rocket name, crew members, equipment, and experiments.

29.8.4. Commit Your Work¶

Be sure to stage and commit your changes!

1. Verify the branch and status of the files.
2. Commit your changes locally.
3. Create a new repository in your GitHub account, then push your commits to origin.

You will make different versions of the mission planning dashboard in the next two studios.

29.8.5. Bonus Mission¶

1. Display crew members by adding an array of crew names.
    1. In `crew.component.ts` add `crew: string[] = ["Jessica Watkins", "Raja Chari", "Jasmin Moghbeli"];`
    2. In `crew.component.html` use references like `<li>{{crew[0]}}</li>` to display the crew names.
2. Use CSS to add different colors, fonts, borders, etc. to your dashboard.
3. Move the components around to see how that affects the display of the data.

- ← 29.7. Exercises: Angular, Lesson 1
- 30. Angular, Part 2 →

Back to top

LaunchCode logo

30. Angular, Part 2¶

30.1. Angular Directives¶

In the DOM chapter you learned how to wait for *events* and then change the appearance of the web page 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.

30.1.1. 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.

Access Angular lesson 2 in VSCode.

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
$ cd lesson2
$ ls
   examples        exercises
$ git branch
   solutions
   * master
```


- ← 30. Angular, Part 2
- 30.2. ngFor →

Back to top

LaunchCode logo

Search

1. Contents
2. 30. Angular, Part 2
3. 30.2. ngFor

30.2. 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.

30.2.1. 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.
    1. The `let` keyword declares the `movie` variable.
    2. `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 `<li>` tag.
4. `{{movie}}` is the placeholder for the current value of `movie`.

By placing the `*ngFor` statement inside the tag, the loop generates multiple `<li></li>` 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.

30.2.2. Try It¶

From the lesson2 folder in VSCode, open the examples/ngfor-practice/src/app/chores folders and select the chores.component.html file.

Access ngFor practice in VSCode.

The starter code should match this:

```
1 <div class='chores'>
2    <h3>Chores To Do Today</h3>
3    <ul>
4       <li>{{chores[0]}}</li>
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.

```
$ pwd
   angular-lc101-projects/lesson2
$ ls
   examples        exercises
$ cd examples
$ ls
   input-practice  ngfor-practice  ngif-practice
$ cd 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:

    1. Replace line 4 with `<li *ngFor = "let chore of chores">{{chore}}</li>`.
    2. Delete lines 5 and 6.
    3. Save your changes.
    4. Reload the web page to verify that all the chores are displayed.

2. Open `chores.component.ts`. Add "Clean bathroom" to the `chores` array, then save. Reload the web page to make sure the new chore appears. Your output should look like this:

    *ngFor first solution.

3. Remove two chores from the array. Reload the web page to make sure these items disappear from the list.

4. Return to `chores.component.html`. Use `*ngFor` within the `<div>` tag to loop over the `todoTitles` array:

    1. Replace line 1 with `<div class='chores' *ngFor = "let title of todoTitles">`.
    2. Replace "Chores To Do Today" in line 2 with a placeholder for `title`.
    3. Save your changes, then reload the page. Properly done, your page should look something like:

    *ngFor practice solution.

5. Return to `chores.component.ts`. Add an item to the `todoTitles` array, then save. Check to make sure another list appears on the web page. Next, remove two items from the `todoTitles` array. Save and make sure the page reflects the changes.

30.2.2.1. 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 `<ul>` tag instead? Try it!

30.2.2.2. Bonus What If¶

What if we want to have different chores listed for Yesterday, Today, and Tomorrow?

*ngFor bonus solution.

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 chores = [
2    {title: "Yesterday's Chores", tasks: ['Empty dishwasher', 'Start
3    {title: "Today's Chores", tasks: ['Load dishwasher', 'Finish Laun
4    {title: "Tomorrow's Chores", tasks: ['Empty dishwasher AGAIN', 'P
5 ]
```

2. Update line 1 in `chores.component.html` to access each *object* in the `chores` array:
    1. `<div class='chores' *ngFor = 'let list of chores'>`
    2. 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'>`.

30.2.3. 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 `<li>` 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 `<li>` 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

- ← 30.1. Angular Directives
- 30.3. `ngIf` →

Back to top

LaunchCode logo

Search

1. Contents
2. 30. Angular, Part 2
3. 30.3. `ngIf`

30.3. `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.

30.3.1. `*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 web page, and the content gets displayed. If `condition` returns `false`, then the HTML tag is NOT generated, and the content stays off the web page.

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:

```
<div class='movies' *ngIf = "movies.length > 3">
   <h3>Movies to Watch</h3>
   <ol>
      <li *ngFor = "let movie of movies">{{movie}}</li>
   </ol>
</div>
```

Adding the `*ngIf` statement inside the `<div>` tag determines whether that element and any content it contains gets added to the web page. 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.

30.3.1.1. 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 web page 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.

30.3.1.2. What About `if/else`?¶

With JavaScript, we know how to use an `if/else` block to decide which set of code to execute:

```
if (num >  5) {
   //Execute this code if num is greater than 5.
} else {
   //Execute this code if num is NOT greater than 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:

```
<someTag *ngIf = "condition; else variableName">
   <!-- HTML tags and content --->
</someTag>

<ng-template #variableName>
   <!-- Alternate HTML tags and content --->
</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`.

```
<div class='movies' *ngIf = "movies.length > 3; else needMoreMovies">
   <h3>Movies to Watch</h3>
   <ol>
      <li *ngFor = "let movie of movies">{{movie}}</li>
   </ol>
</div>

<ng-template #needMoreMovies>
   <div class='movies'>
      <h3>Not Enough Movies!</h3>
      <p>You only have {{movies.length}} movies on your watch list!</p>
      <p>What's up with that?</p>
      <p>You need to purchase expensive popcorn more often.</p>
   </div>
</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 web page. 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.

30.3.2. 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.

```
$ cd ..
$ ls
   input-practice  ngfor-practice  ngif-practice
$ cd ngif-practice
```

In VSCode, open the `chores.component.html` file from the `ngif-practice` folder:

Find the ngIf practice file in VSCode.

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>
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 targetImage 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 web page 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:

*ngIf practice solution.

30.3.3. Check Your Understanding¶

Examine the following code:

```
<div>
   <h3>Prep Work</h3>
   <ul>
      <li *ngFor = "let task of prepWork">{{task}}</li>
   </ul>
</div>

<ng-template #noHW>
   <p>This space intentionally left blank.</p>
</ng-template>
```

Question

Assume we have defined a prepWork array to hold the homework tasks for our next class. We want the web page 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"`

- ← 30.2. ngFor
- 30.4. Events →

Back to top

LaunchCode logo

Search

1. Contents
2. 30. Angular, Part 2
3. 30.4. Events

30.4. Events¶

Recall that *events* are actions that allow users to interact with the content on a web page. 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.

Click handler example.

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.

30.4.1. 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.

30.4.1.1. 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>
   ```


- ← 30.3. ngIf
- 30.5. Responding to User Input →

Back to top

LaunchCode logo

Search

1. Contents
2. 30. Angular, Part 2
3. 30.5. Responding to User Input

30.5. 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.

30.5.1. 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 web page should look like this:

Starting setup for Angular input practice.

Now let's modify the code to accept user input. Be sure to code along with each step.

30.5.2. 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.

Added an input box.

Typing characters into the box does not do anything yet, since we have not included any instructions to deal with the data.

1. 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"/>
```

2. 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 T
   ```

   `(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*.

1. Add a `<p>` element to line 8 of your code, and include a placeholder for the value of `newMovie`. Save and reload the page:

```
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 Mo
8    <p>{{newMovie.value}}</p>
9 </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.

Keyup input practice.

As you add or remove characters from the input box, you should see a real-time update of the text on the screen.

30.5.3. 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.

30.5.3.1. 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'`.

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.

30.5.3.2. Wait for a Click¶

1. 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 T
   ```

   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.

30.5.3.3. 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.

1. 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="E
8    <button (click)='true'>Add</button>
```

```
 9    <p>{{newMovie.value}}</p>
10 </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.

Keyup input practice.

30.5.4. 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 web page.

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.

- ← 30.4. Events
- 30.6. Events Can Call Functions →

Back to top

LaunchCode logo

Search

1. Contents
2. 30. Angular, Part 2
3. 30.6. Events Can Call Functions

30.6. 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(arguments...)"
```

Changing the movie list displayed on the web page 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.

30.6.1. 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='
 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 to the array, we no longer need the title to appear below the input box. Remove `<p>{{newMovie.value}}</p>` from line 9.

30.6.2. Define the Function¶

Open `movie-list.component.ts` and examine the code:

```
 1 import { Component, OnInit } from '@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 MovieListComponent implements OnInit {
 9    movies = ['Toy Story', 'The Shining', 'Sleepless in Seattle', 'The M
10
11    constructor() { }
12
13    ngOnInit() {
14    }
15 }
```

The `movies` array stores the titles displayed on the web page, and we want to update this when the user supplies new information.

1. Declare a function called `addMovie` that takes one parameter:

```
export class MovieListComponent implements OnInit {
   movies = ['Toy Story', 'The Shining', 'Sleepless in Seattle', 'T

   constructor() { }

   ngOnInit() {
   }

   addMovie (newTitle: string) {

   }
}
```

   Notice that we have to declare the data type for the `newTitle` parameter.

2. Now add code to `push` the new title to the `movies` array:

```
export class MovieListComponent implements OnInit {
   movies = ['Toy Story', 'The Shining', 'Sleepless in Seattle', 'T

   constructor() { }

   ngOnInit() {
   }

   addMovie (newTitle: string) {
      this.movies.push(newTitle);
   }
}
```

   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:

Updated movie list.

30.6.3. 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:

Same movie added multiple times.

Let's modify the code to try to prevent this from happening.

30.6.3.1. Clear the Input Box¶

1. 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.val
   ```

   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.

2. 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</b
   ```

   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.

30.6.3.2. 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:

```
addMovie (newTitle: string) {
   if(!this.movies.includes(newTitle)){
      this.movies.push(newTitle);
   }
}
```

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.

30.6.4. 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.

30.6.5. Check Your Understanding¶

Assume that we have an Angular project that presents users with a list of potential pets:

Potential pet list.

Question

Which of the following calls the `addPet` function when the user clicks on one of the potential pets:

1. `<li>{{pet}}</li>`
2. `<li (click) = "true">{{pet}}</li>`
3. `<li #addPet (click) = "true">{{pet}}</li>`
4. `<li (click) = "addPet(pet)">{{pet}}</li>`

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}}</li>`
2. `<li #newFriend (mouseover) = "pet.name">{{pet}}</li>`
3. `<li (mouseover) = "newFriend = pet.name">{{pet}}</li>`
4. `<li (mouseover) = "newFriend">{{pet}}</li>`

- ← 30.5. Responding to User Input
- 30.7. Exercises: Angular, Lesson 2 →

Back to top

30.7. Exercises: Angular, Lesson 2¶

Let's build an interactive web page that allows us to review data for our astronaut candidates and select crew members for a space mission.

30.7.1. 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.

Access lesson 2 exercises in VSCode.

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 web page in your browser, it should look like this:

Starting setup for exercises.

30.7.2. 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 web page.

1. Find the "Candidates" section in `candidates.component.html`. Use `*ngFor` in the `<li>` tag to loop over the `candidates` array and display each name in an ordered list.

   Check your solution.

2. We want each name to be interactive. Add a `click` event to the `<li>` 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:

Candidate results.

30.7.3. 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`.

   Check your solution.

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.

   Check your solution.

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:

Candidate Data results.

30.7.4. 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 `<img>` tag, use `*ngIf` to check if a candidate has been selected.

   Check your solution.

2. Replace the generic `{{placeholder}}` with the `image` property of the candidate.

Properly done, your output should behave something like this:

Sidekick image results.

30.7.5. 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.

   Check your solution.

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.

   Check your solution.

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.

   Check your solution.

30.7.5.1. 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.

   Check your solution.

3. Add a `click` event that clears the `crew` array.

Properly done, your output should behave something like this:

Crew list results.

30.7.6. Bonus Missions¶

30.7.6.1. 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.

30.7.6.2. 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. For this exercise, the `ng-template` code you need is at the bottom of `candidates.component.html`.

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`.

30.7.7. Bonus Results¶

After finishing the bonus missions, your output should behave something like this:

Bonus content behavior.

- ← 30.6. Events Can Call Functions
- 30.8. Studio: Angular, Part 2 →

Back to top

LaunchCode logo

Search

1. Contents
2. 30. Angular, Part 2

30.8. 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.

30.8.1. Getting Started¶

For this studio and the next, you will clone some starter code from GitHub.

1. Fork the Angular Mission Planner repository.
2. In the terminal, move into your root `angular_practice` directory.
3. Create a new subdirectory for the Mission Planner repository
4. `cd` into the new subdirectory. Do NOT `git init`.
5. Clone your fork inside your new subdirectory.

   Warning

   Initializing a new Angular project inside of another one creates version control complications that are best avoided.

   Before running the `git clone` command in the terminal, make sure you are inside the subdirectory you just made! Cloning into a project that already has a repo of some kind will throw an error.

   Using the `ls` command in your terminal is a great way to verify your location.

   Tips for repository organization and init vs clone.
6. Use `git status` to verify that you are on branch `studio-2`. If not, use `git checkout branch-name` to switch to it.

   Warning

   When you fork your repo over, `studio-2` may have been renamed `main` by GitHub. To verify you have the correct code, compare your `crew.component.html` to LaunchCode's `crew.component.html` on the `studio-2` branch.

   You can rename your branches to match the LaunchCode's branch names, or keep it as `main`.

   How to rename a branch (Scroll down to 15.4.5)
7. Run `npm install` to download dependencies.

8. Run `ng serve` to build and serve the project.

30.8.2. 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.

30.8.2.1. 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.

Animated gif of mission name being clicked, edited, and then saved.

Example of mission name being edited.¶

30.8.2.2. 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 { Component, OnInit } from '@angular/core';
 2
 3 @Component({
 4    selector: 'app-crew',
 5    templateUrl: './crew.component.html',
 6    styleUrls: ['./crew.component.css']
 7 })
 8 export class CrewComponent implements OnInit {
 9
10    crew: object[] = [
11       {name: "Eileen Collins", firstMission: false},
12       {name: "Mae Jemison", firstMission: false},
13       {name: "Ellen Ochoa", firstMission: true}
14    ];
15
16    constructor() { }
17
18    ngOnInit() {
19    }
20
21 }
```

30.8.3. Requirements¶

Note

All of these features only *temporarily* alter the data. If you refresh the page, the original data will reappear.

30.8.3.1. 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.

30.8.3.2. Use *ngFor to Display Crew¶

Replace the static list of `<li>` 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>
```

30.8.3.3. Display 1st Mission Status¶

If a crew member's `firstMission` property is `true`, then display the text "- 1st" next to their name.

Example of first mission status appearing next to crew member name.

Example of first mission status being shown.¶

Add this code right after the member name in `src/app/crew/crew.component.html`.

```
1 <span *ngIf="member.firstMission">- 1st</span>
```

30.8.3.4. 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.

Animated gif of crew member being added to list after add button is clicked.

Example of crew member being added.¶

Add this code to the *bottom* of `src/app/crew/crew.component.html`.

```
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 `add` function. This function adds the new crew member to the roster!

In the `src/app/crew/crew.component.ts` file, include this code for the `add` function:

```
1 add(memberName: string, isFirst: boolean) {
2   this.crew.push({name: memberName, firstMission: isFirst});
3 }
```

30.8.3.5. 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 deletes that person from the crew array.

Animated gif of crew member disappearing from the list after the remove button for that item is clicked.

Example of crew member being removed.¶

Add line 4 to file `src/app/crew/crew.component.html`. Be sure to put it before the closing `</li>`, so that the button appears next to each item in the crew list.

```
1 <li *ngFor="let member of crew">
2    {{member.name}}
3    <span *ngIf="member.firstMission">- 1st</span>
4    <button (click)="remove(member)">remove</button>
5 </li>
```

Add the `remove` function shown below to the crew component in the `src/app/crew/crew.component.ts` file.

```
remove(member: object) {
  let index = this.crew.indexOf(member);
  this.crew.splice(index, 1);
}
```

30.8.3.6. 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.

   Animated gif of crew member name being clicked, edited, and then saved.

   Example of crew member name being edited.¶

We need to add a click event to the member name.

1. Put `{{member.name}}` inside of a `<span>` that has a `(click)` handler.
2. Make the `<li>` in `src/app/crew/crew.component.html` look like the code below.

```
<li *ngFor="let member of crew">
  <span (click)="edit(member)" class="editable-text">{{member.name}}
  <span *ngIf="member.firstMission">- 1st</span>
  <button (click)="remove(member)">remove</button>
</li>
```

We need a way of knowing which crew is being edited.

1. 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.

   ```
   memberBeingEdited: object = null;
   ```

2. 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.

```
edit(member: object) {
   this.memberBeingEdited = member;
}
```

Now we need to add an `*ngIf` that will show the two versions of the member, the display state or the edit state.

1. 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.

```
<h3>Crew</h3>
<ul>
   <li *ngFor="let member of crew">

      <span *ngIf="memberBeingEdited !== member; else elseBlock">
         <!-- display state of member -->
         <span (click)="edit(member)" class="editable-text">{{membe
         <span *ngIf="member.firstMission">
            - 1st
         </span>
         <button (click)="remove(member)">remove</button>
      </span>

      <ng-template #elseBlock>
         <!-- edit state of member -->
         <input />
         <button>save</button>
      </ng-template>

   </li>
</ul>
<input #name type="text"/>
<label>First mission<input #firstMission type="checkbox"/></label>
<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.

1. Update the `<input>` and `<button>` tags to look like:

```
<ng-template #elseBlock>
   <!-- edit state of member -->
   <input #updatedName (keyup.enter)="save(updatedName.value, member
   <button (click)="save(updatedName.value, member)">save</button>
</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.

1. Add the below `save` function to the crew component.

```
save(name: string, member: object) {
member['name'] = name;
this.memberBeingEdited = null;
}
```

2. Commit and push up your work.

30.8.4. 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.

30.8.5. Sanity Check¶

Complete code for this studio (without the bonus content) can be found in the `studio-2-solution` branch of the repository.

- ← 30.7. Exercises: Angular, Lesson 2
- 31. Angular, Part 3 →

Back to top

LaunchCode logo

Search

1. Contents
2. 31. Angular, Part 3

31. Angular, Part 3¶

- ← 30.8. Studio: Angular, Part 2
- 31.1. Data Binding →

31.1. Data Binding¶

In the previous two Angular lessons, you *stored* data in a `component.ts` file, and then *displayed* that data in a `component.html` file.

Example

Assign a `listHeading` variable and a `movies` array in the `movie-list.component.ts` file:

```
export class MovieListComponent implements OnInit {
   listHeading: string = "Oscar Winners to Watch";
   movies: string[] = ['Titanic', 'Gladiator', 'Mutiny on the Bounty', '
   constructor() { }
   ngOnInit() { }
}
```

Display the list heading and the titles from the `movies` array using the `movie-list.component.html` file:

```
<div class='movies'>
   <h3>{{listHeading}}</h3>
   <ol>
      <li *ngFor = 'let movie of movies'>{{movie}}</li>
   </ol>
</div>
```

Each Angular component contains an HTML file that organizes the content users see in their web browsers. We call these files the **view**.

Data binding refers to the technique of linking information contained in a file to the view. In the example above, data contained in the `movie-list.component.ts` file is *bound* to placeholders in the `movie-list.component.html` file.

By binding `listHeading` and `movies`, we tell Angular to watch them for changes. Whenever the variables in `movie-list.component.ts` change in value, Angular responds by automatically updating the HTML file.

Data binding is a powerful technique, since it allows developers to focus on the fun part of the code rather than dealing with all the nitty-gritty "ugh-I-need-to-add-statements-here-here-and-here-to-refresh-the-page".

Fortunately, you already experienced setting up data binding in Angular, so the groundwork is done. In this chapter, we will continue practicing the skill, add in the vocabulary, and explore some refinements.

- ← 31. Angular, Part 3
- 31.2. One-Way Data Binding →

Back to top

LaunchCode logo

Search

1. Contents
2. 31. Angular, Part 3
3. 31.2. One-Way Data Binding

31.2. One-Way Data Binding¶

Linking data from a `component.ts` file to the `component.html` file is an example of *one-way* binding. Changes made to the `component.ts` file are reflected in the view, but changes made in `component.html` have no influence on `component.ts`. Information flows in one direction only.

31.2.1. Displaying the Values of Variables¶

As we saw in Angular lessons 1 and 2, the syntax for binding a variable is to encase its name in double braces `{{ }}`. We used this in the exercises and studios to display headings, titles from our movie list, astronauts on our crew, etc.

Example

Inside the `my-list.component.ts` file:

```
1 export class MyListComponent implements OnInit {
2   listHeading: string = 'My Great List';
3   luckyNumber: number = 42;
4
5   constructor() { }
6
```

```
7    ngOnInit() { }
8 }
```

Inside the `my-list.component.html` file:

```
<h3>{{listHeading}}</h3>
<p>{{luckyNumber}}<p>
```

Changing the value of `luckyNumber` in the `MyListComponent` class will change the text displayed by the `p` element. However, altering the HTML will NOT affect the value of `luckyNumber` in the class.

31.2.2. Binding to HTML Attributes¶

We also used one-way data binding to modify attributes within an HTML tag. For example, assume the `image` variable holds the URL for a photo we want to use on our web page.

Example

Inside the `photos.component.ts` file:

```
1 export class PhotosComponent implements OnInit {
2    image: string = 'https://www.launchcode.org/assets/icons/trophy-95e8c
3
4    constructor() { }
5
6    ngOnInit() { }
7 }
```

Inside the `photos.component.html` file:

```
<img src="{{image}}"/>
```

Within the `<img>` tag, we bind the `image` variable to the `src` attribute. Note that the braces and variable name must be inside quotes. When the code runs, the string stored in `image` is used to provide the required URL.

31.2.2.1. Update on the Binding Syntax¶

Angular allows us to use an alternate syntax whenever we use data-binding to modify an HTML attribute.

To avoid cluttering up our HTML tags with lots of double braces `{{ }}`, we simplify the notation as follows:

Examples

Double braces syntax:

```
<img src="{{ variableName }}"/>
<input name="{{ otherVariable }}" value="{{ thirdVariable }}"/>
```

Alternate syntax:

```
<img [src]="variableName"/>
<input [name]="otherVariable" [value]="thirdVariable"/>
```

Instead of {{ }}, place the HTML attribute in square brackets [ ] and set that equal to the variable name in quotes.

> Note: Although the two methods accomplish exactly the same thing, the square brackets syntax is recommended as a best practice.

31.2.3. Check Your Understanding¶

Question

Which of the following is NOT a true statement about data-binding?

1. Data-binding refers to the linking of component information to a view.
2. In one-way binding, information flows in one direction only.
3. In one-way binding, changing a component variable never updates the application view.
4. In one-way binding, user input does not affect component data.

Question

Which of the following show proper data-binding syntax? Choose ALL that apply.

1. `<li>{{ variableName }}</li>`
2. `<img [src]="variableName"/>`
3. `<p>[variableName]</p>`
4. `<button value="[variableName]">Go!</button>`
5. `<input placeholder="{{ variableName }}"/>`
6. `<button [name]="{{ variableName }}"/>`

- ← 31.1. Data Binding
- 31.3. Attribute Directives →

Back to top

LaunchCode logo

Search

1. Contents
2. 31. Angular, Part 3
3. 31.3. Attribute Directives

31.3. Attribute Directives¶

In the previous two chapters, you learned about two of the three Angular directives---components and structural. You will now use data-binding to practice *attribute directives*. These change the appearance of a specific HTML element within the DOM.

31.3.1. Open the Lesson 3 Folder¶

Open VSCode and return to the `angular-lc101-projects` folder. Find `lesson3/examples/src/app` in the sidebar and open the `skill-set.component.ts`, `skill-set.component.html`, and `skill-set.component.css` files.

Access Angular lesson 3 in VSCode.

Open the terminal panel and navigate to the lesson3 `examples` folder. Also make sure that you are on the `master` branch.

```
$ git branch
   * master
   solutions
$ ls
   lesson1 lesson2 lesson3
$ cd lesson3
$ ls
   examples        exercises
$ cd examples
```

Once you are in the folder, enter `npm install` in the terminal, then run `ng serve` to launch the project.

You should see the following:

Attribute directives starting screen.

31.3.2. Update the Skill-Set Styling¶

Examine the code in the `skill-set.component.css` and `skill-set.component.html` files:

Examples

CSS

```
1 h3 {
2   text-align: center;
3   text-decoration: underline;
4 }
```

```
5
6 .skills {
7    color: green;
8 }
```

HTML

```
 1 <div class="skills">
 2    <h3>{{listHeading}}</h3>
 3    <ol>
 4       <li *ngFor="let skill of skills">{{skill}}</li>
 5    </ol>
 6    <hr>
 7    <h3>Copy of Skill List</h3>
 8    <ol>
 9       <li *ngFor="let skill of skills">{{skill}}</li>
10    </ol>
11    <hr>
12    <p>Here is some practice text...</p>
13 </div>
```

Right now, there is an awful lot of green on the page, which is set by the `skills` class in the CSS file. Let's fix this with some attribute directives.

31.3.2.1. Inline Styling¶

To change the color and bullet type of the first list element, we could override the CSS instructions with some inline code:

```
<ol style="color: black" type="A">
```

However, we can use what we learned about data-binding to replace these hard-coded styles with variables:

```
<ol [style.color]="alternateColor" [type]="bulletType">
```

Ideas to note:

1. Unlike the structural directives `*ngFor` and `*ngIf`, we can add more than one attribute directive to an HTML tag.

2. The `style` attribute has different properties that can be assigned using dot notation. Examples include `style.color` and `style.background`. For properties with two-word labels, like `text-align`, the data-binding syntax accepts either hyphens or camel case (`style.text-align` or `style.textAlign`).

3. The variables `alternateColor` and `bulletType` are assigned in the `skill-set.component.ts` file.

```
export class SkillSetComponent implements OnInit {
   listHeading: string = 'Some Coding Skills I Know';
   skills: string[] = ['Loops', 'Conditionals', 'Functions', 'Class
   alternateColor: string = 'black';
   bulletType: string = 'A';
   changeColor: boolean = true;

   constructor() { }

   ngOnInit() { }

}
```

4. NEAT! Reassigning the `alternateColor` variable in the `.ts` file causes EVERY tag with `[style.color]="alternateColor"` to change color.

Try It

Change the values for the `alternateColor` and `bulletType` variables. Save your work and refresh the web page to see the results.

Note that `bulletType` takes options of numbers (`1`), upper and lower case letters (`A`, `a`), or upper and lower case Roman numerals (`I`, `i`). For a detailed description of using the `type` attribute in an ordered list, check out the W3 schools documentation.

31.3.2.2. Changing Styles with Booleans¶

We can accomplish the same results by applying a class to the second `ol` tag:

1. Add the following code to `skill-set.component.css`:

```
.ol-style {
   color: black;
   text-align: center;
   list-style-type: upper-roman;
}
```

   Note

   The CSS property `list-style-type` defines the look of the list item markers, similar to the `ol` element's `type` attribute. The values available to the CSS property are different, however. You can find a full list at W3 schools.

2. Next, modify line 8 in the starter HTML:

```
 1 <div class="skills">
 2    <h3>{{listHeading}}</h3>
 3    <ol [style.color]="alternateColor" [type]="bulletType">
 4       <li *ngFor="let skill of skills">{{skill}}</li>
 5    </ol>
 6    <hr>
 7    <h3>Copy of Skill List</h3>
 8    <ol [class.ol-style]="changeColor">
 9       <li *ngFor="let skill of skills">{{skill}}</li>
10    </ol>
11    <hr>
12    <p>Here is some practice text...</p>
13 </div>
```

After saving these updates, the skills list changes appearance:

Attribute directives midpoint screen.

3. Instead of setting `[class.ol-style]` equal to a string, the `changeColor` variable is a boolean defined in the `skill-set.component.ts` file. If `changeColor` is `true`, Angular adds the `ol-style` class of the tag. If `changeColor` is `false`, the class remains absent from the tag.

Try It

1. Set `changeColor` to `false` and verify that the second ordered list changes back to green, left-aligned, and numbered.
2. Create a `p-style` class in the CSS file and modify line 12 in `skill-set.component.html` to make the color and alignment of the `p` element depend on `!changeColor`.

31.3.3. What About the Buttons?¶

Nice display of eagerness! We will deal with the buttons on the next page.

31.3.4. Check Your Understanding¶

The `reversed` attribute labels ordered lists from highest to lowest values (9, 8, 7... instead of 1, 2, 3...). Unlike the `class` or `style` attributes, `reversed` is not set equal to a string inside the HTML tag. Just having it in the tag flips the numbering of the bullets.

```
<ol style="color: blue" reversed>
```

Question

How could we data-bind the `reversed` attribute in an `ol` tag? Indicate ALL working options.

1. Bind the attribute to a variable that holds the string `"reversed"` or `"notReversed"`.
2. Bind the attribute to a boolean variable set as `true` or `false`.
3. Bind the attribute to a boolean statement like `variable1 > variable2`.
4. Bind the attribute to the empty string `""`.
5. Just put square brackets around `reversed` and hope for the best.

- ← 31.2. One-Way Data Binding
- 31.4. Dynamic Style Changes →

Back to top

LaunchCode logo

Search

1. Contents
2. 31. Angular, Part 3
3. 31.4. Dynamic Style Changes

31.4. Dynamic Style Changes¶

By adding event listeners to our Angular projects, we can make any element on our web page interactive. This allows us to change the styling on the page in response to user actions.

Before we add functionality to the buttons, let's first update our text styling a little bit.

31.4.1. Interactive Elements¶

Let's make the paragraph element respond to user clicks.

Change line 12 in `skill-set.component.html` by adding a `(click)` event:

```
 1 <div class="skills">
 2    <h3>{{listHeading}}</h3>
 3    <ol [style.color]="alternateColor" [type]="bulletType">
 4       <li *ngFor="let skill of skills">{{skill}}</li>
 5    </ol>
 6    <hr>
 7    <h3>Copy of Skill List</h3>
 8    <ol [class.ol-style]="changeColor">
 9       <li *ngFor="let skill of skills">{{skill}}</li>
10    </ol>
11    <hr>
```

```
12    <p [class.p-style]="!changeColor" (click)="changeColor = !changeColo
13 </div>
```

Since `changeColor` is a boolean, `(click)="changeColor = !changeColor"` flips the value of the variable between `true` and `false` whenever the text is clicked.

Interactive text styling.

Notice that since the style of the ordered list also depends on `changeColor`, its appearance changes as well.

Try It

1. Replace `(click)` with `(mouseover)` in line 12 and examine how the interactivity changes.
2. What happens if we add `(click)="changeColor = !changeColor"` to the `<li>` tag in line 9? Does this make a single list element clickable or all of them?

31.4.2. Button Styling¶

The `button` tag accepts `class` and `style` just like other HTML elements, but `button` also has its own set of special attributes. Detailed descriptions of these attributes appear at W3 schools and MDN. However, for these examples we will focus on `disabled`.

Adding the `disabled` attribute inside the tag deactivates the button. Any time you see a greyed out button on a web page, `disabled` is the cause.

Example

Code:

```
1 <button [style.background]="lcLightBlue">Click Me!</button>
2 <button disabled>Can't Click Me!</button>
```

Result:

Result of ``disabled`` attribute.

The `disabled` attribute overrides any other function calls or events within the `button` tag.

31.4.2.1. Dynamic Button Activation¶

Data-binding lets us activate or deactivate buttons based on user actions or variables we control.

Open the `buttons.component.html` and `buttons.component.ts` files and examine the code.

Examples

HTML file:

```
1 <div class="buttons">
2    <h3>{{buttonHeading}}</h3>
3    <button class="gold">Gold</button> <br>
4    <button class="silver">Silver</button> <br>
5    <button class="copper">Copper</button> <hr>
6    <button>Activate/Deactivate Buttons</button>
7 </div>
```

`ButtonsComponent` class in the `.ts` file:

```
1 export class ButtonsComponent implements OnInit {
2    buttonHeading: string = "Buttons"
3    inactive: boolean = false;
4
5    constructor() { }
6
7    ngOnInit() { }
8
9 }
```

To dynamically activate or deactivate a button, we need to bind the attribute to a boolean. In this case, we can use the `inactive` variable defined in the `buttons.component.ts` file.

1. Modify line 3 in the HTML file as follows:

   ```
   <button [class.gold]="!inactive" [disabled]="inactive" (click)="inacti
   ```

   Save your changes and refresh the web page. Now when you click on the "Gold" button, `inactive` gets set to `true`. This adds the `disabled` attribute to the HTML tag, turning it off. In addition, the `gold` class is removed, changing the color of the button.

2. Since the "Gold" button is no longer active, clicking on it again will not trigger any events. To re-enable the button, we need to tie `inactive` to another tag and event.

3. Modify line 6 in the HTML file as follows:

   ```
   <button (click)="inactive = !inactive">Activate/Deactivate Buttons</bu
   ```

The buttons on the page should now behave something like:

Activating and deactivating a button on click.

31.4.2.1.1. Silver and Copper¶

Modify the `button` tags for "Silver" and "Copper" so that they also depend on `inactive`. Properly done, clicking ANY of the buttons deactivates ALL of them:

Activating and deactivating multiple buttons on click.

If we want to disable only the button that gets clicked, then we will need to create a separate boolean variable for each element. We must also update the "Activate/Deactivate" button to reset all of the booleans to `true`. (Hint: How about setting the `(click)` event to a function call?)

31.4.3. Try It!¶

Modify the code in `buttons.component.html` and `buttons.component.ts` to make the buttons behave like this:

Deactivating one button at a time.

Note

If you find yourself stuck after trying, and trying, and TRYING, remember that there is a `solutions` branch in the repository.

31.4.4. Bonus Try It!¶

Just for fun, use what you have learned about events and data binding to create buttons that behave like this:

Fun but frustrating buttons.

Warning

These bonus options are ONLY FOR FUN. Using any of these on a business website would be a poor choice.

31.4.5. Check Your Understanding¶

Question

To include dynamic styles in a component:

1. Only data-binding is needed,
2. Only an event handler is needed,
3. Data-binding and event handling are both necessary,
4. Angular, data-binding, and event handling are all necessary.

Question

Which of the following shows the three Angular directives arranged from the most general to most specific?

1. Components, structural directives, attribute directives
2. Components, attribute directives, structural directives
3. Structural directives, components, attribute directives
4. Structural directives, attribute directives, components
5. Attribute directives, structural directives, components
6. Attribute directives, components, structural directives

Question

Consider the following code samples:

CSS:

```
.special {
   background-color: blue;
}
```

TypeScript:

```
export class Component implements OnInit {

   specialFactor: boolean = false;

   constructor() { }

   ngOnInit() { }

}
```

HTML:

```
<div [class.special]="specialFactor">Very Special Content</div>
```

Which of the following, when added to the HTML tag, will render the `<div>` element blue when clicked?

1. `(click)="!specialFactor"`
2. `(click)="specialFactor = 5 < 3"`
3. `(click)="specialFactor = !specialFactor"`
4. `(click)="specialFactor = false"`

- ← 31.3. Attribute Directives
- 31.5. Angular Wrap-Up →

Back to top

LaunchCode logo

31.5. Angular Wrap-Up¶

Over the past three chapters, you have learned:

1. What *templates* are and why we want to use them,
2. How to use the Angular CLI to create a new project, generate new components, and run a project locally in your web browser.
3. How to define custom HTML tags and use them to arrange components within a template.
4. How to use the *structural directives* `*ngFor` and `*ngIf` to automatically generate HTML elements and display dynamic content.
5. How to catch events and set up data-binding to make your web page respond to user actions.
6. How to use *attribute directives* to modify the appearance and behavior of specific HTML elements.

31.5.1. Tip of the Iceberg¶

The Angular skills you learned are a small slice of what the framework can accomplish. They provide a solid foundation if you choose to learn more on your own.

Of course, practice makes better. The best way for you to get better at Angular is to use Angular. Adapt the examples, exercises, and studio projects however you want. Also, create your own project and build your own sample web pages. Create an interactive family photo page, a grocery list, a web based breakout box, or whatever you want.

You can find full documentation on the Angular.io website. A good "next step" for your Angular learning would be to complete the Tour of Heros tutorial.

Note

The Angular documentation is excruciatingly complete, but do not be intimidated. There are plenty of separate tutorial sites that cover individual skills and techniques. When you find an interesting topic on the Angular site, feel free to Google that topic and explore how other coders explain how to use it.

31.5.1.1. Two Topics to Consider¶

In this chapter, we discussed *one-way* data binding. As the name suggests, *two-way* binding exists as well. We chose not to explore this idea because

one-way tasks are more flexible and more frequently used. However, two-way binding can come in handy if you want data from an `input` element to automatically change a variable in the `.ts` file.

A much larger idea involves passing data between different components in the same template. Passing data from a parent component to its children, from a child to the parent, or between children is VERY useful. Unfortunately, that topic is beyond the scope of these lessons.

Try Googling "Passing data between Angular components", and explore several of the most recent results. Find one that best suits your learning style, and then create a simple template that includes parent-child communication. Also, the *Tour of Heros* tutorial uses component to component communication.

If your future coding job involves Angular, you will most likely need to learn the skill.

31.5.2. Components Should Be Re-used¶

Recall that at their best, functions accomplish only ONE task. This makes them re-usable within the same program, or as a module accessible by many different programs. Components should behave in a similar manner. Each one should do just one simple thing and should be flexible enough to work within many different templates.

The real power of a front end framework comes when you view components as reusable elements. With time and practice, you will build a collection of small, generic components. You can then combine these tools in different ways to build small or large sections of any new project. The Lego analogy works here---the same set of blocks can be combined in multiple ways to produce different outcomes. Also, if you need consistent results on multiple web pages, then you just put the components together the same way each time instead of writing new code.

For demonstration purposes, some of our examples don't use this best practice and instead include components that serve many purposes. After you finish the following exercises and studio, consider how you could split the larger components into smaller, re-usable pieces.

- ← 31.4. Dynamic Style Changes
- 31.6. Exercises: Angular, Part 3 →

Back to top

LaunchCode logo

Search

1. Contents
2. 31. Angular, Part 3
3. 31.6. Exercises: Angular, Part 3

31.6. Exercises: Angular, Part 3¶

Remember that DOM and Events studio? Let's revisit this and do it again Angular-style.

31.6.1. Starter Code¶

The starter code for the exercises is in the same repository that you cloned for the all of the Angular lessons.

When the project is running, it should look like this:

> Initial view of the Angular lesson 3 exercises application in a browser.
>
> Initial view of the exercises app

31.6.2. The Requirements Review¶

The requirements are the same as before:

1. When the "Take off" button is clicked, the following should happen:
    1. A confirmation window should let the user know "Confirm that the shuttle is ready for takeoff." If the shuttle is ready for liftoff, then add steps b-d.
    2. The flight status should change to "Shuttle in flight."
    3. The background color of the shuttle flight screen should change from green to blue.
    4. The shuttle height should increase by 10,000 km.
2. 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.
3. When the "Abort Mission" button is clicked, the following should happen:
    1. A confirmation window should prompt the user if they really want to abort the mission. If the user wants to abort the mission, then add steps b-d.
    2. The flight status should change to "Mission aborted."
    3. The background color of the shuttle flight screen should change from blue to red.
    4. The shuttle height should go down to 0.
4. 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 km.

31.6.3. Add Attribute Directives and Template Variables¶

Open `app.component.ts` and take a look. The `color`, `height`, and `message` properties will all be used as template variables to dynamically update the view.

```
1 export class AppComponent {
2   title = 'Exercises: Angular Lesson 3';
3
4   color = 'green';
5   height = 0;
6   width = 0;
7   message = 'Space shuttle ready for takeoff!';
8 }
```

31.6.3.1. Update the HTML¶

1. Pass in the `message` variable to the app html:

   Check your solution.

2. Add this style directive to line 19: `[style.backgroundColor]="color"`.

3. Use the `height` property to determine the displayed height.

   Check your solution.

4. Refresh the page to ensure your template variables are assigned correctly.

31.6.4. Add Events to Modify Directives¶

31.6.4.1. Control Buttons¶

Now, we'll add some event listeners to the three control buttons on the bottom of the page. These listeners will reassign the values of `color`, `height`, `width`, and `message`.

1. Add an event listener to the *Take Off* button.

   Check your solution.

2. Back in `app.component.ts`, we'll define this listener. The `confirm()` method will look the same as before, but this time we can use a few less lines of code to update the view.

```
handleTakeOff() {
   let result = window.confirm('Are you sure the shuttle is ready fo
   if (result) {
      this.color = 'blue';
      this.height = 10000;
      this.width = 0;
      this.message = 'Shuttle in flight.';
   }
}
```

3. Follow the same pattern to handle the *Land* and *Abort Mission* click events.

31.6.4.2. Movement Buttons¶

Next, we'll tackle the `Up`, `Down`, `Left`, and `Right` buttons that move the rocket. The `movement` formula is the same as we've used before:

```
let movement = parseInt(img.style.left) + 10 + 'px';
```

But now, instead of using the `getElementById` method, we'll access the `img` element by passing it in to the click event.

1. In `app.component.html`, label the `img` element so we can reference it.

   Check your solution.

2. While you're here, add this click handler to the *Right* button: `(click)="moveRocket(rocketImage, 'right')"`

3. Complete the `moveRocket()` method in the app's component class to handle rocket movement to the right:

   Check your solution.

4. Add conditional logic to this `moveRocket()` method to account for the other movement directions, modifying the movement formula as needed. Be sure to also update the `height` or `width` property where appropriate.

31.6.4.3. Update the Control Button Click Handlers¶

Along those same lines, we'll want to modify a couple of our control button handlers to update `rocketImage`'s position when the status changes. Pass in

`rocketImage` to your *Land* and *Abort Mission* handlers and add the following:

```
rocketImage.style.bottom = '0px';
```

31.6.5. New Requirements¶

1. Right now, a user can move the rocket before it has officially taken off or abort the mission while the rocket is still on the ground. This doesn't make much sense. With attribute directives, we can dynamically set those buttons to only be enabled in some states.

   In `app.component.ts`, let's add a check for the take off status of the shuttle.

   Check your solution.

2. When the app is first loaded, we want the user to be able to click the *Take Off* button, but not the *Land* or *Abort Mission* button. In `app.component.html`, update the control buttons to add some `[disabled]` attribute directives.

   Based on the boolean `takeOffEnabled` property, only the *Take Off* control button is enabled when the rocket is on the ground.

   Update the control button click handlers to toggle the enabled/disabled status of the controls using this value.

3. For another improvement, we shouldn't be able to move the rocket if it hasn't taken off. To toggle the status of the direction buttons, we could add more boolean checks to our component. However, we know we only want these buttons to be accessible when the *Take Off* button is not. We can therefore reuse `takeOffEnabled` to determine if the user can click the direction buttons.

   ```
   1 <button (click)="moveRocket(rocketImage, 'up')" [disabled]="takeOffE
   ```

   In fact, since all four direction buttons share the same requirements for disablement, we can take advantage of our old friend `*ngIf` to display the whole set based on `takeOffEnabled`.

   Check your solution.

4. Lastly, let's change the shuttle's background color to a warning color if the rocket image gets too close to the edge. Add a function to your component that will check the width and height values and changes the color value to orange if those values are too high or low. Call that function in each of the direction button click handlers.

31.6.6. Bonus Mission¶

1. Just like the original studio, change the code to prevent the rocket image from flying off the colored background.
2. Dynamically adjust the enabled/disabled status of the direction buttons based on the position of the rocket.

- ← 31.5. Angular Wrap-Up
- 31.7. Studio: Angular, Part 3 →

Back to top

LaunchCode logo

Search

1. Contents
2. 31. Angular, Part 3
3. 31.7. Studio: Angular, Part 3

31.7. Studio: Angular, Part 3¶

In this studio, you must combine data-binding and attribute directives to dynamically change the appearance or behavior of HTML elements.

31.7.1. Getting Started¶

This studio uses the same mission planner repository as Angular studio part 2. There is no need to fork and clone new starter code.

1. Open your work from Part 2 in Visual Studio Code.
2. Run `git status` to see if you have any uncommitted work. If you do, resolve it.
3. Checkout the `studio-3` branch: `git checkout studio-3`.
4. Run `npm install` to download dependencies.
5. Run `ng serve` to build and serve the project.

When viewed in your browser, the project should look like this:

Initial view of Angular studio 3.

Mission Name and *Carrier Rocket* are still editable, but the functionality for the crew, equipment, and experiments have been removed.

31.7.2. Part 1: Select Cargo¶

We want to add items from the equipment list to the cargo hold, but we must NOT exceed the values of the `maximumAllowedMass` or `maxItems` variables.

The interactive equipment list will eventually behave as follows:

1. When clicked, an *Add to Cargo Hold* button adds the selected equipment to the `cargoHold` array and updates the `cargoMass` variable.
2. As items are added to the hold, their names should appear in the Cargo Hold section of the page. Also, the *Mass in Hold*, *Mass Budget Remaining*, and *Spots Filled* values should update.
3. The *Add to Cargo Hold* buttons should be disabled if all of the spots in the hold have been filled.
4. If the mass of a particular item will push the cargo hold over `maximumAllowedMass`, that item's button should be disabled.
5. If `cargoMass` comes within 200 kg of `maximumAllowedMass`, then the *Mass Budget Remaining* text should turn red.

The following sections will guide you through completing Part 1.

31.7.2.1. Code the `addItem` Function¶

Open `equipment.component.ts` and create the `addItem` function:

1. It should take an equipment object as a parameter.
2. It should add the equipment object to the `cargoHold` array.
3. It should increase the `cargoMass` variable by the mass of the new equipment.
4. It should return `true` or `false` depending on whether `cargoMass` is within 200 kg of `maximumAllowedMass`.

31.7.2.2. Make the `Add to Cargo Hold` Buttons Work¶

Open `equipment.component.html` and do the following:

1. Include an *Add to Cargo Hold* button within each list item.

   Minimal *Add to Cargo Hold* button style.

2. Add a `(click)` event to each button that calls the `addItem` function and passes the selected equipment as the argument.

3. Bind the `disabled` attribute to the following conditions:

   1. If all of the cargo hold spots are full (`cargoHold.length === maxItems`), disable the button.
   2. If adding the item to the cargo hold would exceed `maximumAllowedMass`, disable the button.

4. If active, make the button an attractive color.

Final *Add to Cargo Hold* button style.

31.7.2.3. Update the `Cargo Hold` Display¶

Open `equipment.component.html` and `equipment.component.css` and do the following:

1. Add a `nearMaxMass` class in the CSS file that styles the text to be:
    1. Bold OR italic,
    2. Red.
2. Bind `[class.nearMaxMass]` to a boolean that will change the style of the *Mass Budget Remaining* text whenever the cargo hold gets within 200 kg of `maximumAllowedMass`.
3. Add an *Empty Hold* button that clears the `cargoHold` array and resets `cargoMass`. As a side effect, clearing the hold should reactivate all of the buttons and return *Mass Budget Remaining* to its original style.

31.7.2.4. Status Check¶

At this point, the equipment component should behave something like:

Gif of Equipment list and Cargo Hold interactivity.

Before moving on to part 2, be sure to commit and push your work.

31.7.3. Part 2: Select Crew Members¶

We want to add up to three astronauts to the mission crew, and we want to do this by clicking on their names rather than creating more buttons.

The interactive candidates list will eventually behave as follows:

1. When clicked, the candidate's name will change color and will appear in the *Crew* list.
2. If a candidate is already part of the crew, clicking their name again in the *Candidates* list will remove them from the crew.
3. When the mouse pointer hovers over an astronaut's name in the *Crew* list, their photo appears below the list. When the pointer leaves their name, their photo disappears.
4. When the crew size reaches 3 members, the heading changes to *Crew Full* and clicking on more candidate names will not do anything.

The following sections will guide you through completing Part 2.

31.7.3.1. Code the `addCrewMember` Function¶

Open `crew.component.ts` and create the `addCrewMember` function:

1. It should take a candidate object as a parameter.
2. It should check if the candidate is already part of the crew.
3. If the crew size is less than 3 AND the candidate is not part of the crew, then their data should be added to the `crew` array.

4. If the candidate is already part of the crew, then their data should be removed from the `crew` array.

31.7.3.2. Update the Candidates List¶

Open `crew.component.html` and `crew.component.css` and do the following:

1. Add a `(click)` event to each `li` element that calls the `addCrewMember` function and passes the selected candidate as the argument.
2. Add a `selected` class in the CSS file that styles the text to be a different color from the other list items.
3. Bind `[class.selected]` to a boolean statement that will change the color of a candidate's name when they are selected or de-selected for the crew.

31.7.3.3. Update the Crew List¶

1. When the crew size reaches 3, the heading should change to "Crew Full".
2. Add `(mouseover)` and `(mouseout)` events to the `li` tags to determine if the mouse pointer is currently over a name in the *Crew* list.
3. If a crew member is selected by moving the mouse over their name:
    1. Use an `img` tag with `*ngIf` to display a photo of the astronaut below the crew list.
    2. Bind the `.photo` property of the astronaut to the `src` attribute.
    3. When the mouse pointer moves off of a name, the photo should disappear.

31.7.3.4. Status Check¶

At this point, the crew component should behave something like:

Gif of the Candidates and Crew list interactivity.

Before moving on to the bonus missions, be sure to commit and push your work.

31.7.4. Bonus Missions¶

To boost your Angular skills, add one or more of the following features:

1. Update the CSS files to make the web page look a little less bland.
2. Don't allow more than two of the same item in the cargo hold.
3. Allow the user to remove individual items from the hold.
4. Complete the `experiments` component with features similar to the `crew` and `equipment` components.
5. Add other data to the astronaut objects, and center this data below the crew photo.

31.7.5. Sanity Check¶

Complete code for this studio can be found in the `studio-3-solution` branch of the repository.

- ← 31.6. Exercises: Angular, Part 3
- 32. Booster Rockets →

Back to top

LaunchCode logo

Search

1. Contents
2. 32. Booster Rockets

32. Booster Rockets¶

- ← 31.7. Studio: Angular, Part 3
- 32.1. You Got This →

Back to top

LaunchCode logo

Search

1. Contents
2. 32. Booster Rockets
3. 32.1. You Got This

32.1. You Got This¶

> If you can’t fly then run, if you can’t run then walk, if you can’t walk then crawl, but whatever you do, you have to keep moving forward.
>
> —Martin Luther King Jr.

A wise programmer once told me that the reassuring thing about coding is that there is always an answer. She said that while much of life is uncertain and we're faced with questions that don't have definite answers, when we program we can be certain that there is an answer to whatever problem that we have. We just need to find it.

This is where persistence or *frustration tolerance* comes in. What characterizes people who succeed as computer programmers is not how much math they know, or whether they took apart computers when they were eleven years old, or even whether they have a computer science degree. It is how hard they will work to find a solution.

What it takes to succeed as you go through this course and beyond in your coding journey is a determination to not give up. To keep trying new things. To keep taking another look at the problem and coming up with another idea for how to solve it. To accept that this maddening process is not some flaw in you or in the field. But rather that it is the nature of the job itself. That you can learn to love it. And that's why you will earn the big bucks.

Use the following pages when ever you need some motivation/inspiration, i.e., a boost.

Remember,

> Rome wasn't built in a day.

- ← 32. Booster Rockets
- 32.2. Best Practices: Learning To Code →

Back to top

LaunchCode logo

Search

1. Contents
2. 32. Booster Rockets
3. 32.2. Best Practices: Learning To Code

32.2. Best Practices: Learning To Code¶

32.2.1. Traits of Good Programmers¶

Curious. Good communicator. Creative. Detail-oriented. Persistent. Problem solver.

These are some traits that successful computer programmers share. People come to computer programming from all different walks of life and previous job experiences. But the things they have in common are that they like to solve problems. They can both imagine (or invent) the "big picture" as well as pay attention to the "little details". They can express their ideas and listen attentively to the ideas of others. Finally, they are persistent--they don't give up!

As you'll soon discover, coding is not easy. It is interesting and often fun (especially as you get more experienced in it), but it is rarely easy. You'll often run into difficulties that you need to work through and solve more or less on your own. So while we'd like to say that "anyone can learn to code", it is more accurate to say that "anyone who is willing to work hard and persist through difficulties can code."

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*.

32.2.2. Practice Makes Perfect¶

> 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.

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?

1. The student did not understand how to solve the problem.
2. The student knew how to solve the problem and decided to skip the task.
3. The student tried to solve the problem, could not get the program to work, so deleted the code.
4. 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:

1. Even if you have no clue how to approach a task, MAKE AN ATTEMPT ANYWAY, then ask questions.
2. 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.
3. 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.
4. 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.

32.2.3. 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.

32.2.4. Best Practices¶

Here are some 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.
8. DO make a running list of things you've learned and watch it grow each class.

And don't forget:

> Learning takes work, and you need the practice. **Do your homework**.

- ← 32.1. You Got This
- 32.3. The Power of Persistence →

Back to top

LaunchCode logo

Search

1. Contents
2. 32. Booster Rockets
3. 32.3. The Power of Persistence

32.3. The Power of Persistence¶

32.3.1. Embrace Failure¶

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.

32.3.2. Where Would We Be?¶

What if Thomas Edison had given up?

Thomas Edison's teachers said he was too stupid to learn anything. He was fired from his first two jobs for being non-productive. As an inventor, Edison made 1,000 unsuccessful attempts at inventing the light bulb.

When a reporter asked, "*How did it feel to fail 1,000 times?*"
Edison replied, "*I didn't fail 1,000 times. The light bulb was an invention with 1,000 steps.*"

Or if Steve Jobs had taken Hewlett-Packard's advice and finished college?

Apple Computer founder Steve Jobs on attempts to get companies interested in his and Steve Wozniak's personal computer:

So we went to Atari and said, "Hey, we've got this amazing thing, even built with some of your parts, and what do you think about funding us? Or we'll give it to you. We just want to do it. Pay our salary, we'll come work for you." And they said, 'No.'

So then we went to Hewlett-Packard,
and they said, 'Hey, we don't need you. You haven't got through college yet.'

> Do…or do not. There is no try. **Do your homework**.

- ← 32.2. Best Practices: Learning To Code
- 32.4. Let's Play Ball! →

Back to top

LaunchCode logo

Search

1. Contents
2. 32. Booster Rockets
3. 32.4. Let's Play Ball!

32.4. Let's Play Ball!¶

Attention sports fans! Embrace your favorite quote(s):

Sport	Quote
Baseball	

Sport	Quote
	"There may be people who have more talent than you, but there's no excuse for anyone to work harder than you do." - *Derek Jeter*
Basketball	"I've missed more than 9000 shots in my career. I've lost almost 300 games. 26 times I've been trusted to take the game winning shot ... and missed. I've failed over and over and over again in my life. That is why I succeed." - *Michael Jordan*
Gymnastics	"I'd rather regret the risks that didn't work out than the chances I didn't take at all." - *Simone Biles*
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." - *Jerry Rice*
Soccer	"The backbone of success is...hard work, determination, good planning, and perseverance." - *Mia Hamm*
Tennis	"If I don't get it right, I don't stop until I do." - *Serena Williams*
Rocky	How can you listen to this and NOT be inspired? (Gonna Fly now)
Optional	Imagine your favorite motivational phrase here. - *Some admired person*

32.4.1. 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:

1. Begin by getting good at running 1 mile.
2. Then get good at running 3 miles.
3. 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.
4. 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.
5. 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:

1. Begin with "Hello world!"
2. Then learn variables, strings and arrays.
3. Then learn if/else statements and loops, followed by functions and modules.
4. Then code your first "half-marathon".

5. Continue practicing to increase your skills. You WILL earn that marathon.js sticker as you build solid demo projects and complete more interviews.
6. Welcome, fellow coder. Don't forget to inspire others.

And don't forget:

> Your heros worked really hard, so should you. **Do your homework**.

- ← 32.3. The Power of Persistence
- 32.5. Brain Breaks →

Back to top

LaunchCode logo

Search

1. Contents
2. 32. Booster Rockets
3. 32.5. Brain Breaks

32.5. Brain Breaks¶

via GIPHY

32.5.1. Step Away From Your Code¶

Sometimes you just need to take a break. When you find yourself getting frustrated it's time to do something else for a bit and come back with fresh eyes.

Exercise, go for a walk, do the dishes, put on some music and dance, meditate, set a timer and look up memes (bonus share them with your peers). Does not matter what you do. By giving your brain a break, often times when you come back to the code you'll have new insight to the problem that had frustrated you in the first place.

32.5.2. Pick Me Ups¶

The following are curated lists from fellow LaunchCoders. By no means are they definitive, but rather a jumping off point.

Music

Make your own playlist, or search for a tech/it/coding/homework music compilations on your internet player of choice (i.e., pandora, spotify, youtube). Some LaunchCode favorites to get you started:

Gloria by Laura Branigan
Gonna Fly Now (Theme from 'Rocky')
I Will Survive by Gloria Gaynor
Happy by Pharrell Williams
Journey - Don't Stop Believin'
Stronger (What Doesn't Kill You) by Kelly Clarkson
For Once In My Life by Stevie Wonder
Girl On Fire by Alicia Keys
Rise Up by Andra Day
Roar by Katy Perry
Fight Song by Rachel Platten
I'm Still Standing by Elton John
Guardians of the Galaxy: Awesome Mix (Vol. 1 & Vol. 2)
Respect by Aretha Franklin

Ted Talks

Visit TED.Com to view the entire library of TED Talks. Below are some of LaunchCoders favorites.

Grit: The power of passion and perseverance by Angela Lee Duckworth!
Start With Why by Simon Sinek
Achieve your goals with practice, persistence and patience by Ryan Roxie
Inside the mind of a master procrastinator by Tim Urban
or if reading is more your thing Blog Post Master Procrastinator
Learning how to learn by Barbara Oakley
The power of vulnerability by Brene Brown
Nature. Beauty. Gratitude. by Louie Schwartzberg
How to make stress your friend by Kelly McGonigal
A virtual choir 2,000 voices strong by Eric Whitacre
The surprising habits of original thinkers by Adam Grant

Books

The Alchemist by Paulo Coelho
Peak: Secrets from the New Science of Expertise by Anders Ericsson and Robert Pool.
A Mind for Numbers: How to Excel at Math and Science (Even If You Flunked Algebra) by Barbara Oakley
Getting Things Done by David Allen

Misc

Excellent article, Screw motivation, what you need is discipline.
* Note that there are a few expletives used in it, so if you are sensitive to swearing, you may not wish to read it.

Wizard Zines by Julia Evans
Bukola's Youtube Channel on Life as a Developer

And remember,

You Got This!

- ← 32.4. Let's Play Ball!
- Index →

Back to top

'