

1

Course syllabus for Introduction to Front-End Development

This course is the first of a series that aims to help you learn more about web development and prepares you for using Bootstrap on a biographical page you will create. By the end of this course, you'll be able to:

- Describe the front-end developer role
- Explain the core and underlying technologies that power the internet
- Use HTML to create a simple webpage
- Use CSS to control the appearance of a simple webpage
- Explain what React is
- Describe the applications and characteristics of the most popular UI frameworks

In this course, you will explore the following:

Module 1: Get started with web development

In this module, you are introduced to web development. You'll learn about the different types of web developer roles and the responsibilities of front-end, back-end and full-stack developers. You will get a streamlined overview of the core technologies of HTML, CSS and JavaScript and explore the concepts that underpin how the internet works. Furthermore, you will be able to access hands-on exercises to edit a website.

After completing this module, you will be able to:

1. Describe the web developer job role.
2. Distinguish between front-end, back-end and full-stack developers.
3. Explain how data moves through the internet.
4. Describe the technologies that underpin the internet.

Module 2: Introduction to HTML5 and CSS

Here you'll learn about HTML5 and CSS. You'll also examine how to construct HTML documents and add basic styling and layout using CSS.

After completing this module, you will be able to:

1. Use HTML to create a simple webpage.
2. Use CSS to define the style of a simple webpage.

Module 3: UI Frameworks

In this module, you'll learn about UI frameworks. In addition, you will learn how to use the Bootstrap framework to build responsive interfaces. You'll explore the benefits of working with UI frameworks.

After completing this module, you will be able to:

1. Outline the concepts that exist in most UI frameworks.
2. Use the Bootstrap CSS framework to create webpages.
3. Leverage Bootstrap documentation to reproduce and modify CSS components.
4. Use Bootstrap themes.
5. Describe the basics of React in relation to other frameworks and web technologies.

Module 4: Graded Assessment

Here you'll learn about the graded assessment. After you complete the individual units in this module, you'll synthesize the skills from the course to create and style a biographical page. You'll also have the opportunity to reflect on the course content and the learning path that lies ahead.

After completing this module, you will be able to:

1. Create and style a biographical page.

Completed

2

How to be successful in this course

Taking an online course can be overwhelming. How do you learn at your own pace and successfully achieve your goals?

Here are some general tips that can help you stay focused and on track.

Set daily goals for studying

Ask yourself what you hope to accomplish in your course each day. Setting a clear goal can help you stay motivated and beat procrastination. The goal should be specific and easy to measure, such as "I'll watch all the videos in Module 2 and complete the first programming assignment". And don't forget to reward yourself when you make progress towards your goal!

Create a dedicated study space

It's easier to recall information if you're in the same place where you first learned it, so having a dedicated space at home to take online courses can make your learning more effective. Remove any distractions from the space and if possible, make it separate from your bed or sofa. A clear distinction between where you study and where you take breaks can help you focus.

Schedule time to study on your calendar

Open your calendar and choose a predictable, reliable time that you can dedicate to watching lectures and completing assignments. This helps ensure that your courses won't become the last thing on your to-do list.

Tip: You can add deadlines for a Coursera course to your Google calendar, Apple calendar, or another calendar app.

Keep yourself accountable

Tell your friends about the courses you're taking, post achievements to your social media accounts or blog about your homework assignments. Having a community and support network of friends and family to cheer you on makes a difference!

Actively take notes

Taking notes can promote active thinking, boost comprehension and extend your attention span. It's a good strategy to internalize knowledge whether you're learning online or in the classroom. So, grab a notebook or find a digital app that works best for you and start synthesizing key points.

Tip: While watching a lecture on Coursera, you can click the 'Save Note' button below the video to save a screenshot to your course notes and add your own comments.

Join the discussion

Course discussion forums are a great place to ask questions about assignments, discuss topics, share resources and make friends. Our research shows that learners who participate in the discussion forums are 37% more likely to complete a course. So make a post today!

Do one thing at a time

Multitasking is less productive than focusing on a single task at a time. Researchers from Stanford University found that "People who are regularly bombarded with several streams of electronic information cannot pay attention, recall information or switch from one job to another as well as those who complete one task at a time." Stay focused on one thing at a time. You'll absorb more information and complete assignments with greater productivity and ease than if you were trying to do many things at once.

Take breaks

Resting your brain after learning is critical to high performance. If you find yourself working on a challenging problem without much progress for an hour, take a break. Walking outside, taking a shower or talking with a friend can help you to re-energize and even give you new ideas on how to tackle the project.

Your learning journey starts now!

While preparing for the module quiz or working on achieving your learning goals you're encouraged to:

- Work through each lesson in the learning pathway. Try not to skip any activities or lessons unless you are certain that you already know this information well enough to move ahead.
- Take the opportunity to go back and watch a video or read all the information provided before moving on to the next lesson or module.
- Complete all the knowledge and module quizzes and exercises.
- Read the feedback carefully when answering quizzes, as this will help you to reinforce what you are learning.
- Make use of the practical learning environment provided by the exercises. You can gain substantial reinforcement of your learning through the step-by-step application of your skills.

Completed

Capstone project demo: Little Lemon website

You've just begun your coding journey on the Meta front-end developer program. By the end of this program, you'll put your new skills to work by completing a real-world portfolio project, where you'll create your own dynamic front-end web application. Completing this project will help you to validate the knowledge and skills that you have gained.

What you will be able to develop

You will build this web app yourself in React and use all of the excellent tools available. Putting your newly acquired skills into practice, you will demonstrate how to build and program part of a responsive web app.

It includes the following elements:

- A home screen with information about the restaurant.
- A table reservation system.
- A profile screen for users to enter their personal details.
- Navigation that enables users to move between parts of the web app.

Project preview

In the video below, you can take a trip into the future and get a preview of what you'll be able to create with your new set of skills at the end of this program.

Play Video

That's what you'll be able to accomplish by the end of this Front-end development program. Pretty cool, right? It's a modern front-end application for the Little Lemon restaurant and the best thing about this project is that it's part of your portfolio and you'll be able to showcase it to any prospective employer as proof of your abilities. It shows the multiple skills you will learn during this program.

Conclusion

In this reading, you explored the front-end application for the Little Lemon restaurant that demonstrates the kind of project you will develop towards the end of the program.

Best of luck in your coding journey.

Completed

Additional Resources

Learn more

Here is a list of resources that may be helpful as you continue your learning journey.

What is a Web Server? (NGINX)

<https://www.nginx.com/resources/glossary/web-server/>

What is a Web Browser? (Mozilla)

<https://www.mozilla.org/en-US/firefox/browsers/what-is-a-browser/>

Who invented the Internet? And why? (Kurzgesagt)

<https://youtu.be/21eFwbb48sE>

What is Cloud Computing? (Amazon)

<https://youtu.be/mxT233EdY5c>

Browser Engines (Wikipedia)

https://en.wikipedia.org/wiki/Browser_engine

Completed

HTTP examples

This reading explores the contents of HTTP requests and responses in more depth.

Request Line

Every HTTP request begins with the request line.

This consists of the HTTP method, the requested resource and the HTTP protocol version.

`GET /home.html HTTP/1.1`

In this example, `GET` is the HTTP method, `/home.html` is the resource requested and HTTP 1.1 is the protocol used.

HTTP Methods

HTTP methods indicate the action that the client wishes to perform on the web server resource.

Common HTTP methods are:

HTTP Method	Description
-------------	-------------

GET	The client requests a resource on the web server.
-----	---

POST	The client submits data to a resource on the web server.
------	--

PUT	The client replaces a resource on the web server.
-----	---

DELETE	The client deletes a resource on the web server.
--------	--

HTTP Request Headers

After the request line, the HTTP headers are followed by a line break.

There are various possibilities when including an HTTP header in the HTTP request. A header is a case-insensitive name followed by a colon and then followed by a value.

Common headers are:

1

2

3

4

5

Host: example.com

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0)
Gecko/20100101 Firefox/50.0

Accept: */*

Accept-Language: en

Content-type: text/json

- The **Host** header specifies the host of the server and indicates where the resource is requested from.
- The **User-Agent** header informs the web server of the application that is making the request. It often includes the operating system (Windows, Mac, Linux), version and application vendor.
- The **Accept** header informs the web server what type of content the client will accept as the response.
- The **Accept-Language** header indicates the language and optionally the locale that the client prefers.
- The **Content-type** header indicates the type of content being transmitted in the request body.

HTTP Request Body

HTTP requests can optionally include a request body. A request body is often included when using the HTTP POST and PUT methods to transmit data.

1

2

3

4

5

6

7

8

POST /users HTTP/1.1

Host: example.com

{

 "key1": "value1",

 "key2": "value2",

 "array1": ["value3", "value4"]

}

1

2

3

4

5

PUT /users/1 HTTP/1.1

Host: example.com

Content-type: text/json

{"key1": "value1"}

HTTP Responses

When the web server is finished processing the HTTP request, it will send back an HTTP response. The first line of the response is the status line. This line shows the client if the request was successful or if an error occurred.

HTTP/1.1 200 OK

The line begins with the HTTP protocol version, followed by the status code and a reason phrase. The reason phrase is a textual representation of the status code.

HTTP Status Codes

The first digit of an HTTP status code indicates the category of the response: Information, Successful, Redirection, Client Error or Server Error.

The common status codes you'll encounter for each category are:

1XX Informational

Status Code	Reason Phrase	Description
100	Continue	The server received the request headers and should continue to send the request body.
101	Switching Protocols	The client has requested the server to switch protocols and the server has agreed to do so.

2XX Successful

Status Code	Reason Phrase	Description
200	OK	Standard response returned by the server to indicate it successfully processed the request.
201	Created	The server successfully processed the request and a resource was created.
202	Accepted	The server accepted the request for processing but the processing has not yet been completed.
204	No Content	The server successfully processed the request but is not returning any content.

3XX Redirection

Status Code	Reason Phrase	Description
301	Moved Permanently	This request and all future requests should be sent to the returned location.
302	Found	This request should be sent to the returned location.

4XX Client Error

Status Code	Reason Phrase	Description
400	Bad Request	The server cannot process the request due to a client error, e.g., invalid request or transmitted data is too large.
401	Unauthorized	The client making the request is unauthorized and should authenticate.
403	Forbidden	The request was valid but the server is refusing to process it. This is usually returned due to the client having insufficient permissions for the website, e.g., requesting an administrator action but the user is not an administrator.
404	Not Found	The server did not find the requested resource.
405	Method Not Allowed	The web server does not support the HTTP method used.

5XX Server Error

Status Code	Reason Phrase	Description
500	Internal Server Error	A generic error status code given when an unexpected error or condition occurred while processing the request.
502	Bad Gateway	The web server received an invalid response from the Application Server.
503	Service Unavailable	The web server cannot process the request.

HTTP Response Headers

Following the status line, there are optional HTTP response headers followed by a line break. Similar to the request headers, there are many possible HTTP headers that can be included in the HTTP response.

Common response headers are:

2

3

4

Date: Fri, 11 Feb 2022 15:00:00 GMT+2

Server: Apache/2.2.14 (Linux)

Content-Length: 84

Content-Type: text/html

- The **Date** header specifies the date and time the HTTP response was generated.
- The **Server** header describes the web server software used to generate the response.
- The **Content-Length** header describes the length of the response.
- The **Content-Type** header describes the media type of the resource returned (e.g. HTML document, image, video).

HTTP Response Body

Following the HTTP response headers is the HTTP response body. This is the main content of the HTTP response.

This can contain images, video, HTML documents and other media types.

1

2

3

4

5

6

7

8

9

HTTP/1.1 200 OK

Date: Fri, 11 Feb 2022 15:00:00 GMT+2

Server: Apache/2.2.14 (Linux)

Content-Length: 84

Content-Type: text/html

<html>

<head><title>Test</title></head>

<body>Test HTML page.</body>

</html>

Completed

Other Internet Protocols

Hypertext Transfer Protocols (HTTP) are used on top of Transmission Control Protocol (TCP) to transfer webpages and other content from websites. This reading explores other protocols commonly used on the Internet.

Dynamic Host Configuration Protocol (DHCP)

You've learned that computers need IP addresses to communicate with each other. When your computer connects to a network, the Dynamic Host Configuration Protocol or DHCP as it is commonly known, is used to assign your computer an IP address. Your computer communicates over User Datagram Protocol (UDP) using the protocol with a type of server called a DHCP server. The server keeps track of computers on the network and their IP addresses. It will assign your computer an IP address and respond over the protocol to let it know which IP address to use. Once your computer has an IP address, it can communicate with other computers on the network.

Domain Name System Protocol (DNS)

Your computer needs a way to know with which IP address to communicate when you visit a website in your web browser, for example, meta.com. The Domain Name System Protocol, commonly

known as DNS, provides this function. Your computer then checks with the DNS server associated with the domain name and then returns the correct IP address.

Internet Message Access Protocol (IMAP)

Do you check your emails on your mobile or tablet device? Or maybe you use an email application on your computer? Your device needs a way to download emails and manage your mailbox on the server storing your emails. This is the purpose of the Internet Message Access Protocol or IMAP.

Simple Mail Transfer Protocol (SMTP)

Now that your emails are on your device, you need a way to send emails. The Simple Mail Transfer Protocol, or SMTP, is used. It allows email clients to submit emails for sending via an SMTP server. You can also use it to receive emails from an email client, but IMAP is more commonly used.

Post Office Protocol (POP)

The Post Office Protocol (POP) is an older protocol used to download emails to an email client. The main difference in using POP instead of IMAP is that POP will delete the emails on the server once they have been downloaded to your local device. Although it is no longer commonly used in email clients, developers often use it to implement email automation as it is a more straightforward protocol than IMAP.

File Transfer Protocol (FTP)

When running your websites and web applications on the Internet, you'll need a way to transfer the files from your local computer to the server they'll run on. The standard protocol used for this is the File Transfer Protocol or FTP. FTP allows you to list, send, receive and delete files on a server. Your server must run an FTP Server and you will need an FTP Client on your local machine. You'll learn more about these in a later course.

Secure Shell Protocol (SSH)

When you start working with servers, you'll also need a way to log in and interact with the computer remotely. The most common method of doing this is using the Secure Shell Protocol, commonly referred to as SSH. Using an SSH client allows you to connect to an SSH server running on a server to perform commands on the remote computer. All data sent over SSH is encrypted. This means that third parties cannot understand the data transmitted. Only the sending and receiving computers can understand the data.

SSH File Transfer Protocol (SFTP)

The data is transmitted insecurely when using the File Transfer Protocol. This means that third parties may understand the data that you are sending. This is not right if you transmit company files such as software and databases. To solve this, the SSH File Transfer Protocol, alternatively called the Secure File Transfer Protocol, can be used to transfer files over the SSH protocol. This ensures that the data is transmitted securely. Most FTP clients also support the SFTP protocol.

Completed

Exercise: Examine a web page

Introduction

In this exercise, you will practice examining an HTML page using the developer tools.

Goal

- Inspect the HTML document using the developer tools in your browser.

Objectives

- Find the HTML ID of the Little Lemon logo.

Instructions

Step 1: Download the following file on your local system.

[examine_the_page](#)

[ZIP File](#)

Step 2: Unzip the file.

[On Windows](#), open your Downloads folder, right-click the file `examine_the_page.zip` and select Extract All.

X

←  Extract Compressed (Zipped) Folders

Select a Destination and Extract Files

Files will be extracted to this folder:

C:\examine_the_page\

[Browse...](#)

Show extracted files when complete

[Extract](#)

[Cancel](#)

On Mac, open your Downloads folder and double click the file *examine_the_page.zip*.

Once unzipped, there will be a folder named *examine_the_page*.

Step 3: Open the folder and double click *index.html* to view the file in your local web browser. Verify that it looks like this:

LITTLE LEMON

Our Menu

Our restauraunt will be closed on New Year's Day

Falafel New

Chickpea, herbs, spices.

Fried Calamari

Squid, buttermilk.

Pasta Salad

Lettuce, vegetables, mozzarella.

Greek Salad

Cucumbers, onion, feta cheese.

Order Online

Step 4: Right-click the Little Lemon logo and select Inspect (or Inspect Element)

LITTLE LEMON

Our Me

Our restauraunt will be closed on New Year's Day

Inspect

Step 5: Inspect the line in the HTML for the logo in the developer tools panel. The line begins with .

Note: In the line, there is an ID in the following format id="???". Note the value that the ID is equal to.

```
<html>
  > <head>...</head>
  > <body>
    >> <div class="container">
      >>> <div class="row" style="flex">
        >>>> <div class="col-12">
          >>>>> <div class="text-center">
            >>>>>>  == $0
          >>>>></div>
        >>>></div>
      >>></div>
    >>></div>
    >>> <div class="row" style="flex">
      >>>> <div class="row" style="flex">
        >>>>> <div class="row" style="flex">
          >>>>>> <script src="bootstrap.bundle.min.js"></script>
        >>>>></div>
      >>>></div>
    >>></div>
  >>></body>

<html>
  > <head>...</head>
  > <body>
    >> <div class="container">
      >>> <div class="row" style="flex">
        >>>> <div class="col-12">
          >>>>> <div class="text-center">
            >>>>>>  == $0
          >>>>></div>
        >>>></div>
      >>></div>
    >>></div>
    >>> <div class="row" style="flex">
      >>>> <div class="row" style="flex">
        >>>>> <div class="row" style="flex">
          >>>>>> <script src="bootstrap.bundle.min.js"></script>
        >>>>></div>
      >>>></div>
    >>></div>
  >>></body>
```

Tips

- If you get stuck, close the developer tools and start over.
- Review the lesson *Developer Tools*.

Completed

Exercise: Edit a website using a browser developer tools

Introduction

In this exercise, you will practice editing an HTML page using the developer tools.

Goal

- Edit the HTML document using the developer tools in your browser.

Objectives

- Change the text of Our Menu to Little Lemon Menu.

Instructions

Step 1: Download the following file on your local system.

Note: If you have completed the exercise Examine the Page, you can skip steps 1 and 2 as the file contains the same assets from that exercise.

[examine_the_page](#)

[ZIP File](#)

Step 2: Unzip the file.

On Windows, open your Downloads folder, right-click the file `examine_the_page.zip` and select Extract All.

X

←  Extract Compressed (Zipped) Folders

Select a Destination and Extract Files

Files will be extracted to this folder:

C:\examine_the_page\

[Browse...](#)

Show extracted files when complete

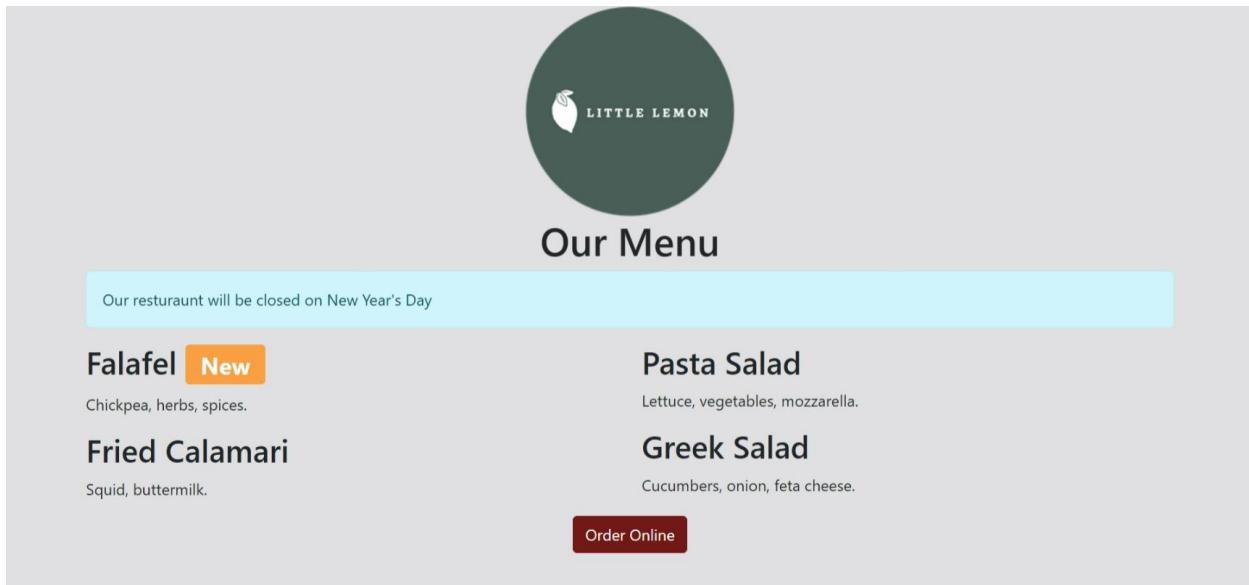
[Extract](#)

[Cancel](#)

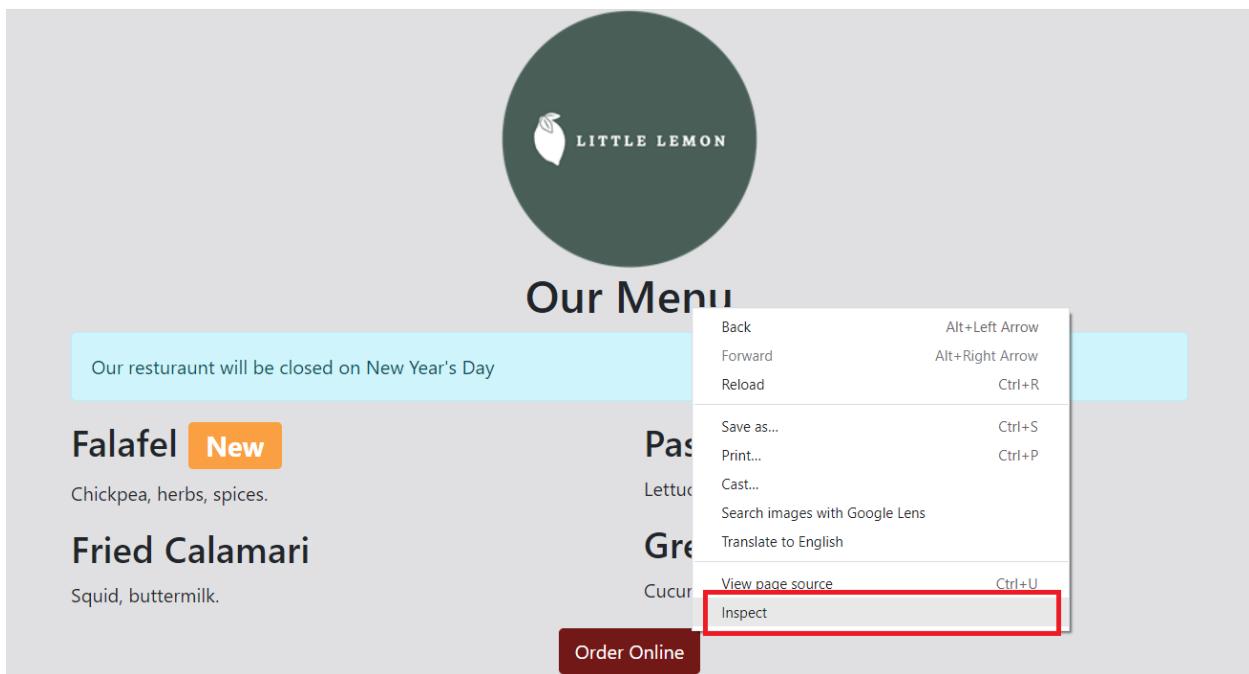
On Mac, open your Downloads folder and double click the file *examine_the_page.zip*.

Once unzipped, there will be a folder named *examine_the_page*.

Step 3: Open the index.html file in your local browser for preview. Verify that it looks like this:



Step 4: Right-click the `Our Menu` text and select `Inspect` or `Inspect Element`.



Step 5: Double-click the `our Menu` text in the Elements tab of the developer tools panel.

Step 6: Change the text to `Little Lemon Menu`.

Step 7: Close the developer tools.

Step 8: Verify that the text has changed on the web page.

Tips

- If you get stuck, close the developer tools and start from the beginning.
- Review the lesson *Developer Tools*.

Completed

Setting up your local development environment

This reading walks you through the steps to set up an Integrated Development Environment, or IDE, on a Windows and on a Mac (further down below).

The IDE you'll be using in the course is Visual Studio Code, which Microsoft provides for free and comes with a wealth of plugins and extensions to make your life as a developer easier.

You have two options for using Visual Studio Code to complete your course activities:

Option 1: Use Visual Studio Code in-browser with Coursera Labs

Coursera's platform offers an in-browser version of Visual Studio Code which is preconfigured and requires no local setup. You can access the Visual Studio Code environment through the "Lab" items included in this course. Your work and files will be saved and available within this in-browser lab while you have course access.

Option 2: Work on your local device

You can also choose to complete your work on your local machine if you prefer. This will require a few steps of set up in advance.

First, you need to download the IDE from Microsoft's website -

<https://code.visualstudio.com/download>.

Select the download based on your operating system.

Windows

Step 1: Download the Windows installer.

Step 2: Open the file to install it once the download is complete.

Step 3: Review and accept the license agreement, then click Next.

**License Agreement**

Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

This license applies to the Visual Studio Code product. Source Code for Visual Studio Code is available at <https://github.com/Microsoft/vscode> under the MIT license agreement at <https://github.com/microsoft/vscode/blob/main/LICENSE.txt>. Additional license information can be found in our FAQ at <https://code.visualstudio.com/docs/supporting/faq>.

MICROSOFT SOFTWARE LICENSE TERMS**MICROSOFT VISUAL STUDIO CODE**

- I accept the agreement
 I do not accept the agreement

Next >

Cancel

Step 4: Keep the default value when prompted for the destination location and click next.



Setup - Microsoft Visual Studio Code (User)



Select Destination Location

Where should Visual Studio Code be installed?



Setup will install Visual Studio Code into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

 Browse...

At least 292.3 MB of free disk space is required.

< Back

Next >

Cancel

Step 5: On the additional tasks view, make sure that **Add to PATH** is selected.

**Select Additional Tasks**

Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Visual Studio Code, then click Next.

Additional icons:

-
- Create a desktop icon

Other:

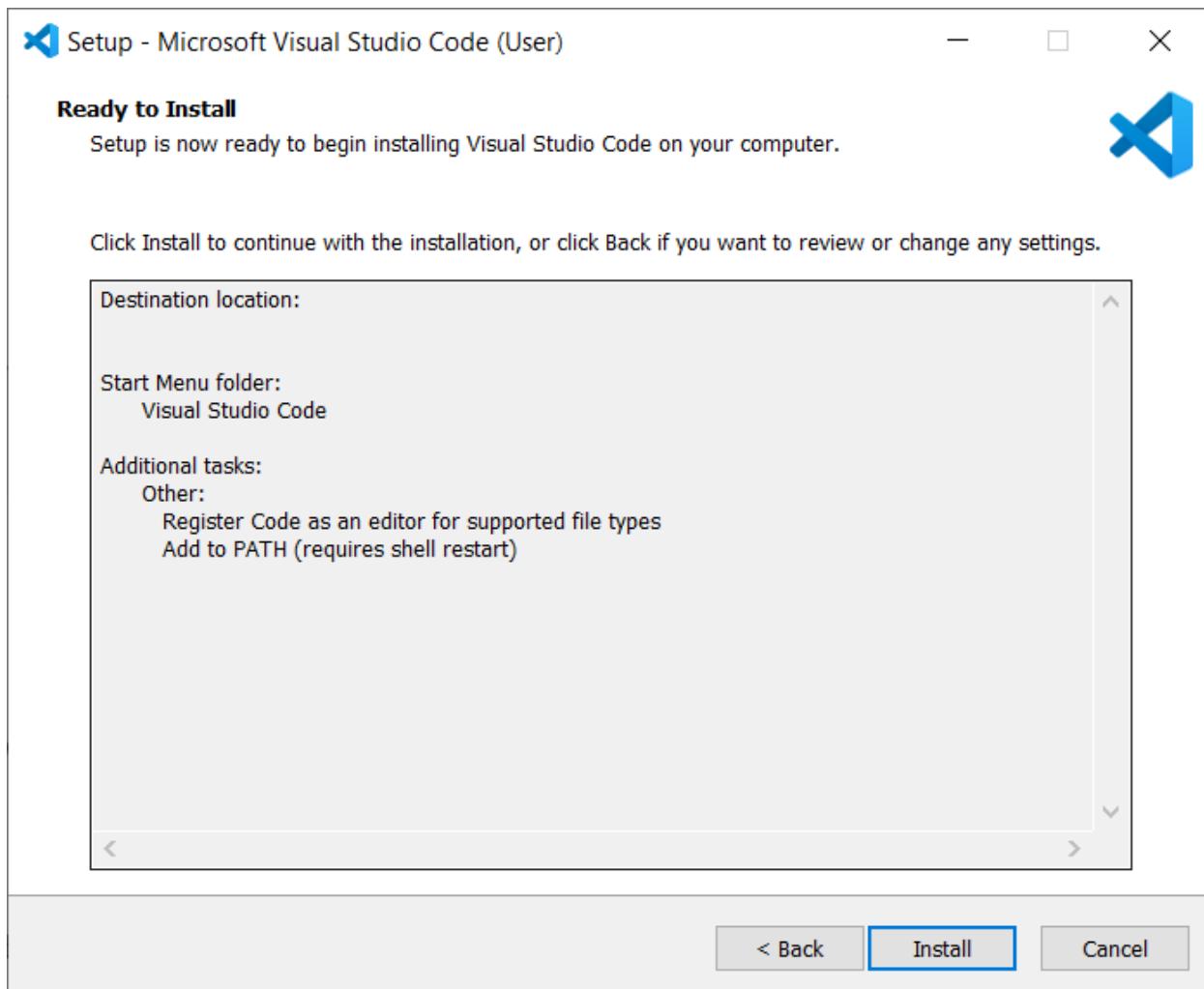
- Add "Open with Code" action to Windows Explorer file context menu
- Add "Open with Code" action to Windows Explorer directory context menu
- Register Code as an editor for supported file types
- Add to PATH (requires shell restart)

< Back

Next >

Cancel

Step 6: Click next.**Step 7:** Click install when the ready to install page appears.



Step 8: Click finish once completed, and the application will launch.

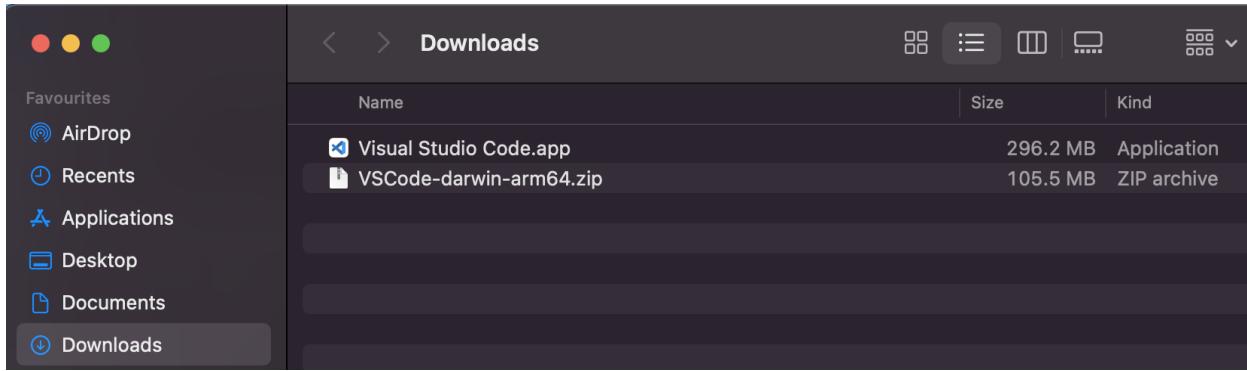
Mac

Step 1: Download the application based on the chipset you have. M1 macs use Apple Silicon, and older Macs use Intel. If you are not sure, choose the Universal option.

Step 2: Go to your Downloads folder once the download is complete.

Step 3: Double-click the zip file to extract the contents.

Step 4: Drag and drop the .app file to the application link in Finder below.



Step 5: Open the app.

Linux

Please refer to the [official Linux installation guide](#) for Visual Studio Code.

Selecting a working directory

Now that you have Visual Studio Code set up create a folder on your device that you'll use to do course exercises.

Open Visual Studio Code, go to `File` and select `Open Folder`. Using the file browser, select the folder you just created.

Congratulations, you're set up now to begin writing some code.

Completed

Visual Studio Code on Coursera

In addition to having Visual Studio Code installed on your own computer, in this course and throughout this program, you'll have the opportunity to work in Visual Studio Code right here on Coursera!

As you progress through the course, you'll be able to write code in hands-on activities called **Labs**. In these labs you'll be able to open Visual Studio Code and start writing code without ever leaving the course.

You'll have plenty of opportunities to see Labs in action later in the course, but for now, use the images below as a visual guide to how Labs will look and operate in your browser.

Lab: Creating an HTML Document

The Labs contain instructions explaining the coding task.

Creating an HTML Document

[Open Lab](#) ↗

Instructions



Introduction

In this ungraded lab you will practice creating a simple HTML document.

Goal

- Create a valid HTML document that displays a piece of text.

Objectives

- Add the DOCTYPE.
- Add the HTML, head and body elements.
- Add the title element.
- Add the text to the body element.

Instructions

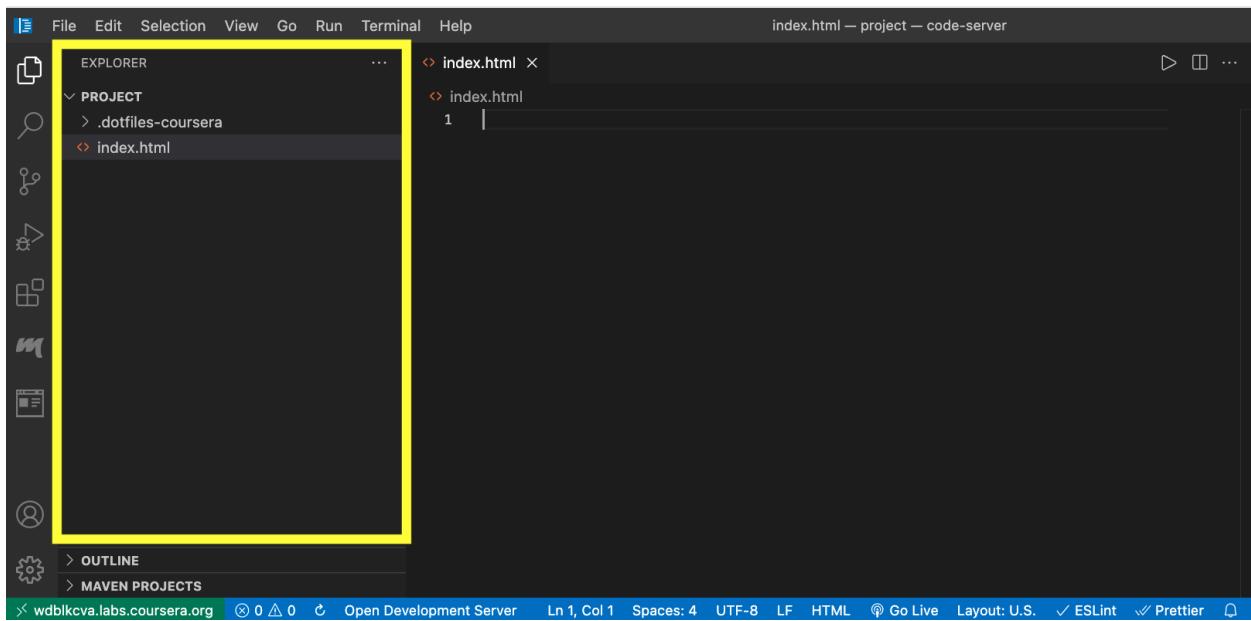
When you click the button to open the lab, a new tab will open with Visual Studio Code already setup and ready for you to start writing code!



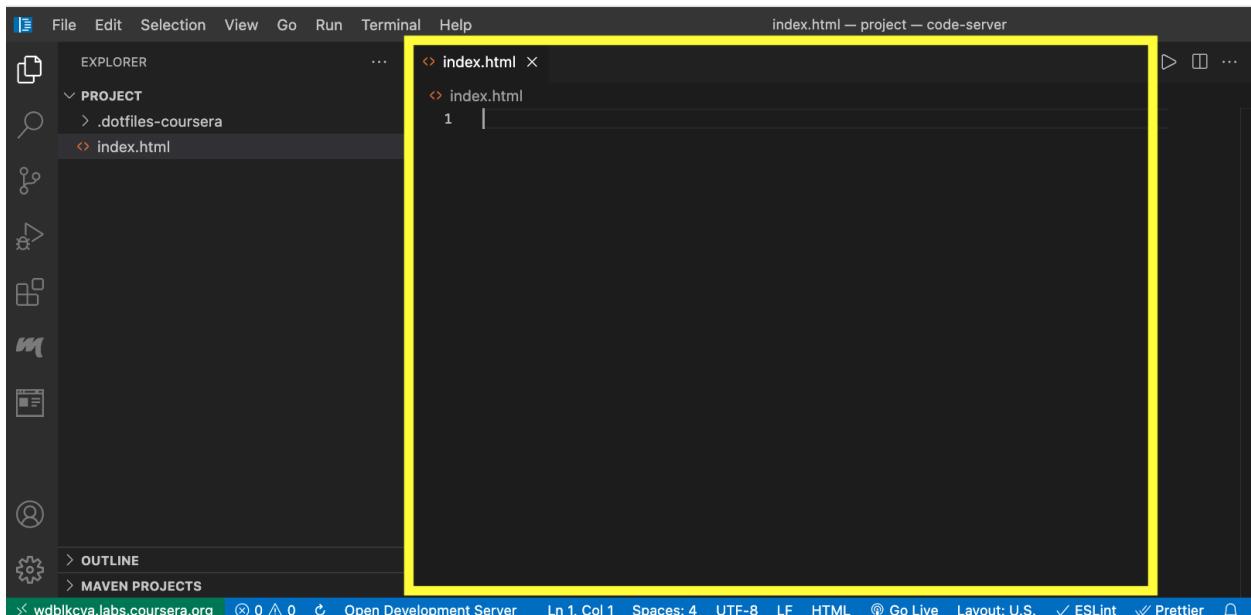
[Navigate](#) [Lab Files](#) [Help](#)

The screenshot shows the Visual Studio Code interface. The top bar includes the Coursera logo, file navigation, and tabs for 'Navigate', 'Lab Files', and 'Help'. The left sidebar has icons for Explorer, Search, and others. The 'EXPLORER' view shows a 'PROJECT' folder containing '.dotfiles-coursera' and 'index.html'. The main editor area shows the 'index.html' file with the text '1'. The bottom status bar displays the URL 'wdblcvva.labs.coursera.org', file statistics (0△0), an open development server, and various code analysis tools like ESLint and Prettier.

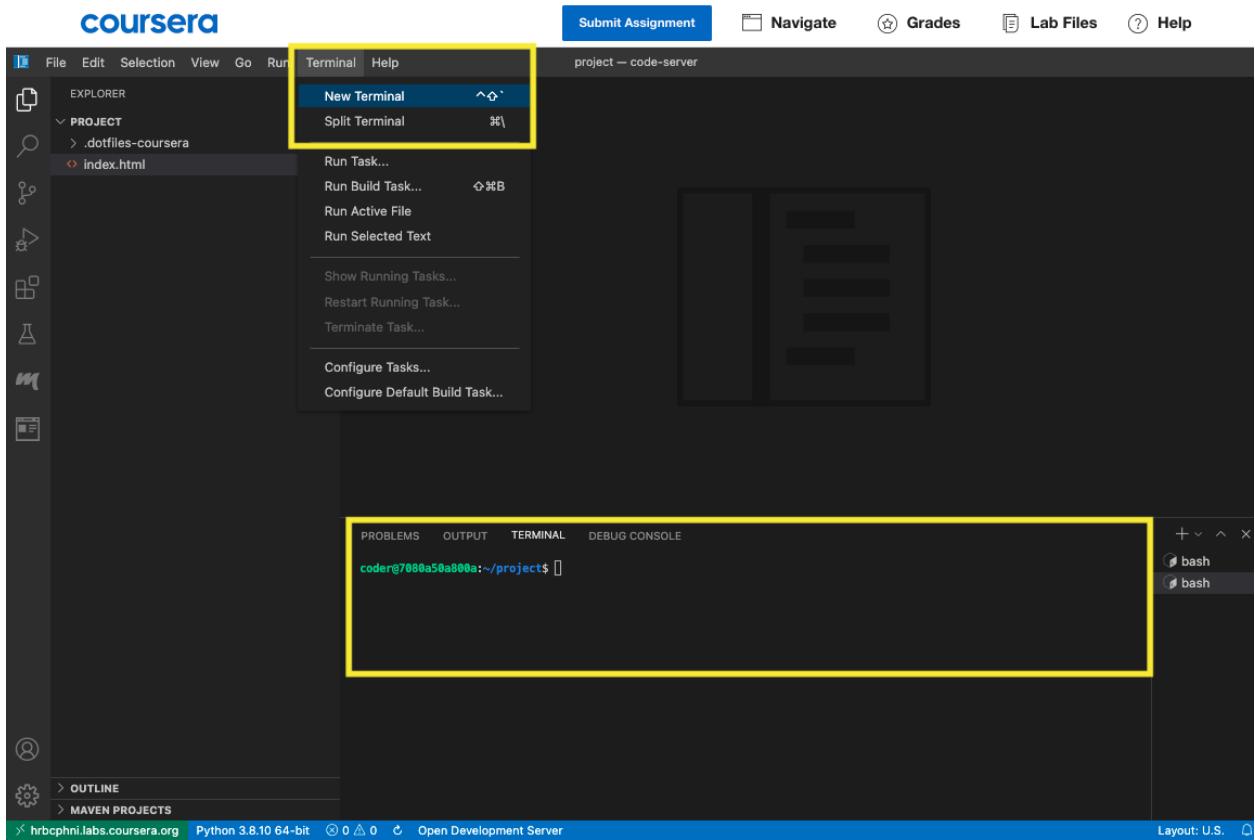
You'll see all the files for the lab in the Project folder in the left sidebar.

[Navigate](#)[Lab Files](#)[Help](#)

And the large editor area where you write your code for the lab.

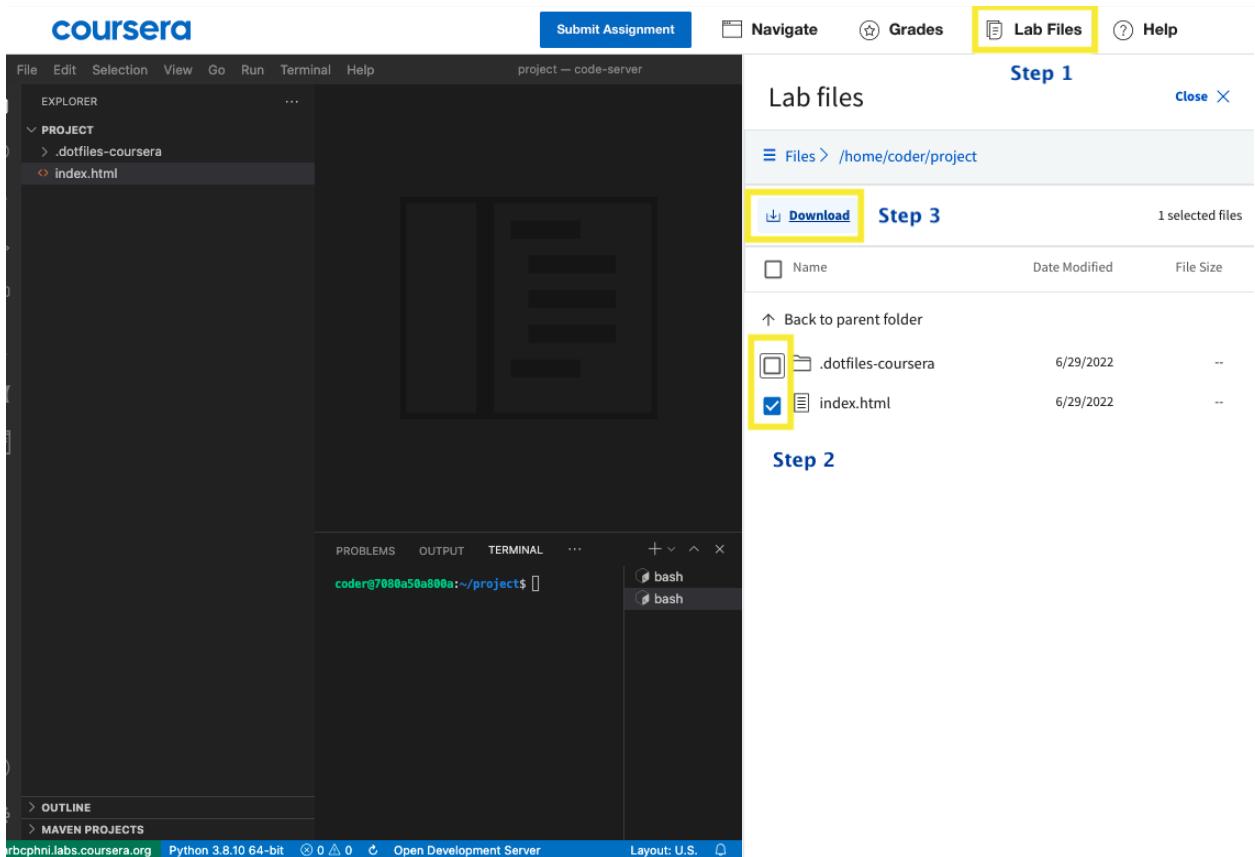
[Navigate](#)[Lab Files](#)[Help](#)

You may need to use a tool called the Terminal from time to time to complete course activities. You can open this by selecting the **Terminal** option in the upper Visual Studio Code toolbar.



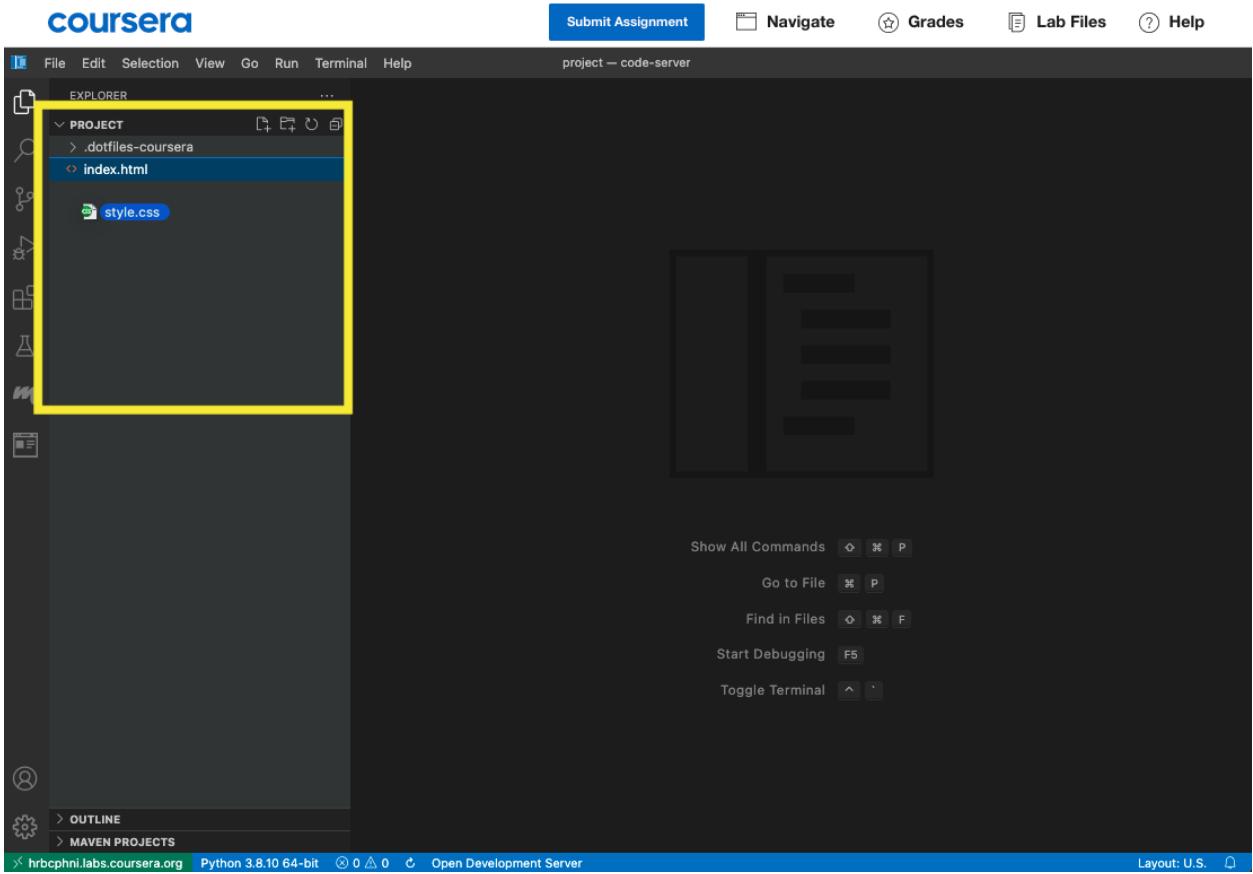
How to download files from your Visual Studio Code Lab to your local device

1. Select the **Lab Files** button in your Lab Toolbar.
2. You'll be able to download your full workspace, specific folders, or individual files through the checkbox selection tool.
3. After you've selected these files, use the **Download** link to download your files to your local device.



How to upload local files to your Visual Studio Code Lab

If you'd like to upload your course files from your local device to your Visual Studio Code lab, **drag and drop** your file from your local device into the Visual Studio Code file tree.



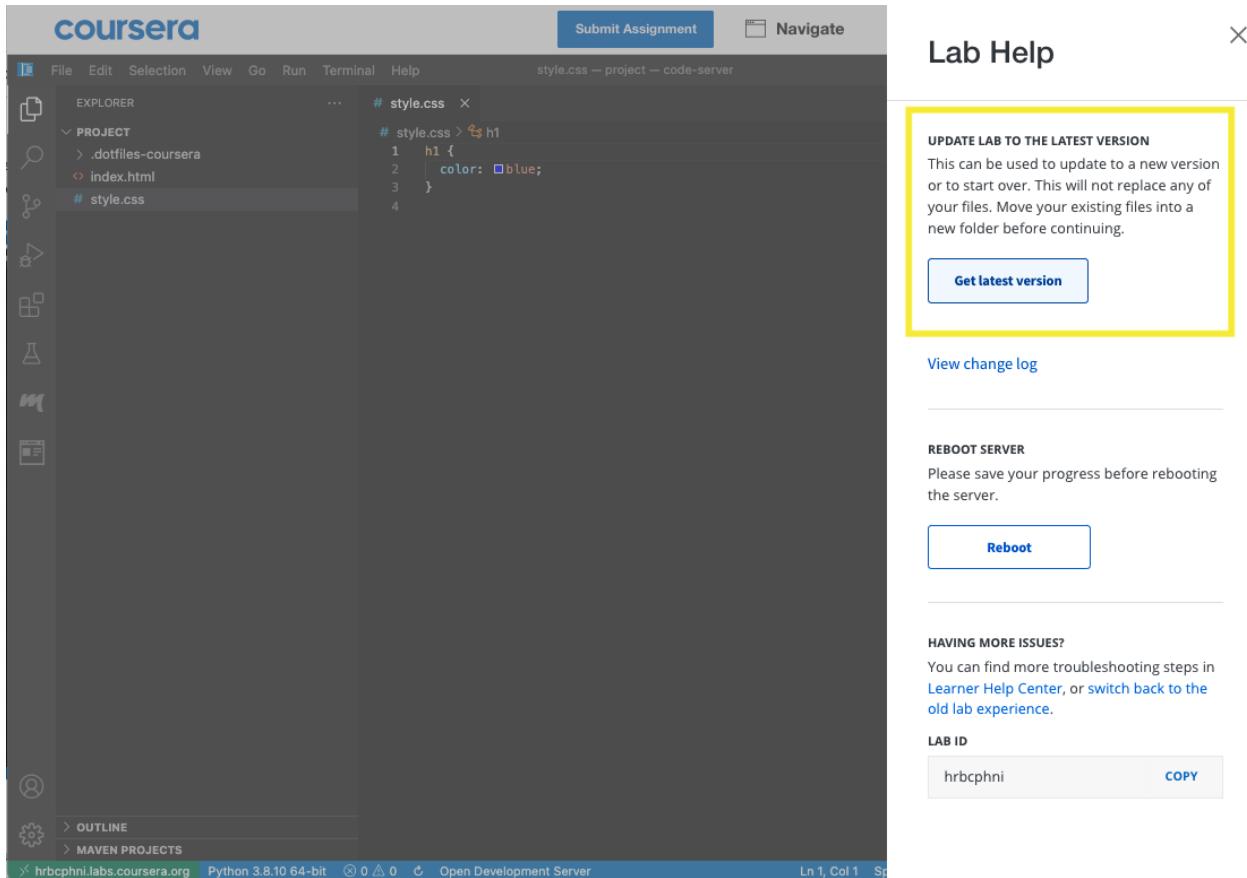
How to get a fresh copy of course-provided starter files

Your work will be saved and persist within your Visual Studio Code lab while you are enrolled in the course. If you'd like to get a fresh copy of the original instructor-provided files at any time, you can do this through the **Lab Help** option in your Lab Toolbar. Don't worry - your original work and files will still remain in your lab until you personally remove or delete them, even when refreshing your files through the steps below.

1. First rename your original files to something like `[yourfilename] [original].[your file extension]`. You can do this by right-clicking on your file in the Visual Studio Code file tree, selecting **Rename**, and providing a new file name.

- For example for `index.html`, this could be renamed to `'index [original].html'`

2. Select **Lab Help** from your Lab Toolbar and then select **Get latest version**.



3. You should now see a fresh copy of the original instructor-provided files in your lab, in addition to your own (renamed) files.

Completed

Additional Resources

Learn more Here is a list of resources that may be helpful as you continue your learning journey.

HTTP Overview (Mozilla)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

Introduction to Networking by Dr.Charles R Severance

<https://www.amazon.com/Introduction-Networking-How-Internet-Works/dp/1511654945/>

Chrome Developer Tools Overview (Google)

<https://developer.chrome.com/docs/devtools/overview/>

Firefox Developer Tools User Docs (Mozilla)

<https://firefox-source-docs.mozilla.org/devtools-user/index.html>

Getting Started with Visual Studio Code (Microsoft)

<https://code.visualstudio.com/docs>

Completed

Simple HTML tags

There are many tags available in HTML. Here you will learn about common tags that you'll use as a developer.

Headings

Headings allow you to display titles and subtitles on your webpage.

1

2

3

4

5

6

7

8

```
<body>  
  <h1>Heading 1</h1>  
  
  <h2>Heading 2</h2>  
  
  <h3>Heading 3</h3>  
  
  <h4>Heading 4</h4>  
  
  <h5>Heading 5</h5>  
  
  <h6>Heading 6</h6>  
  
</body>
```

The following displays in the web browser:

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

Paragraphs

Paragraphs contain text content.

1

2

3

4

5

< p >

This paragraph

contains a lot of lines

but they are ignored.

< / p >

The following displays in the web browser:

This paragraph contains a lot of lines but they are ignored.

Note that putting content on a new line is ignored by the web browser.

Line Breaks

As you've learned, line breaks in the paragraph tag line are ignored by HTML. Instead, they must be specified using the `
` tag. The `
` tag does not need a closing tag.

1

2

3

4

5

```
<p>
```

```
    This paragraph<br>
```

```
        contains a lot of lines<br>
```

```
            and they are displayed.
```

```
</p>
```

The following displays in the web browser:

```
This paragraph  
contains a lot of lines  
and they are displayed.
```

Strong

Strong tags can be used to indicate that a range of text has importance.

1

2

3

<p>

No matter how much the dog barks: <**strong**>don't feed him chocolate</**strong**>.

</p>

The following displays in the web browser:

No matter how much the dog barks: **don't feed him chocolate.**

Bold

Bold tags can be used to draw the reader's attention to a range of text.

1

2

3

<p>

The primary colors are <**b**>red</**b**>, <**b**>yellow</**b**> and <**b**>blue</**b**>.

</p>

The following displays in the web browser:

The primary colors are **red, yellow and blue.**

The following displays in the web browser:

In case of emergency, **push the red button.**

Bold tags should be used to draw attention but not to indicate that something is more important. Consider the following example:

1

The three core technologies of the Internet are **HTML**, **CSS** and **Javascript**.

The following displays in the web browser:

The three core technologies of the Internet are **HTML, CSS and Javascript**.

Emphasis

Emphasis tags can be used to add emphasis to text.

1

2

3

<p>

Wake up **now!**

</p>

The following displays in the web browser:

Wake up *now!*

Italics

Italics tags can be used to offset a range of text.

1

2

3

<p>

The term **HTML** stands for HyperText Markup Language.

</p>

The following displays in the web browser:

The term *HTML* stands for HyperText Markup Language.

Emphasis vs. Italics

By default both tags will have the same visual effect in the web browser. The only difference is the meaning.

Emphasis tags stress the text contained in them. Let's explore the following example:

[1](#)

I <**em**>really</**em**> want ice cream.

The following displays in the web browser:

I *really* want ice cream.

Italics represent off-set text and should be used for technical terms, titles, a thought or a phrase from another language, for example:

[1](#)

My favourite book is <**i**>Dracula</**i**>.

The following displays in the web browser:

My favourite book is *Dracula*.

Screen readers will not announce any difference if an *italics* tag is used.

Lists

You can add lists to your web pages. There are two types of lists in HTML.

Lists can be unordered using the <**ul**> tag. List items are specified using the <**li**> tag, for example:

[1](#)

[2](#)

[3](#)

4

5

6

```
<ul>  
  <li>Tea</li>  
  <li>Sugar</li>  
  <li>Milk</li>  
</ul>
```

This displays in the web browser as:

- Tea
- Sugar
- Milk

Lists can also be ordered using the `` tag. Again, list items are specified using the `` tag.

1

2

3

4

5

```
<ol>  
  <li>Rocky</li>  
  <li>Rocky II</li>  
  <li>Rocky III</li>
```

```
</ol>
```

This displays as the following in the web browser.

1. Rocky
2. Rocky II
3. Rocky III

Div tags

A `<div>` tag defines a content division in a HTML document. It acts as a generic container and has no effect on the content unless it is styled by CSS.

The following example shows a `<div>` element that contains a paragraph element:

```
1<div>  
2  <p>This is a paragraph inside a div</p>  
3</div>
```

This displays as the following in the web browser.

This is a paragraph inside a div

It can be nested inside other elements, for example:

```
1<div>  
2  <div>  
3    <p>This is a paragraph inside a div</p>  
4  </div>  
5</div>
```

```
<div>
```

```
<div>  
  <p>This is a paragraph inside a div that's inside another div</p>  
  </div>  
</div>
```

This displays in the web browser as:

This is a paragraph inside a div that's inside another div

As mentioned, the div has no impact on content unless it is styled by CSS. Let's add a small CSS rule that styles all divs on the page.

Don't worry about the meaning of the CSS just yet, you'll explore CSS further in a later lesson. In summary, you're applying a rule that adds a border and some visual spacing to the element.

1

2

3

4

5

6

7

8

9

10

11

```
<style>
```

```
  div {
```

```
border: 1px solid black;  
  
padding: 2px;  
  
}  
  
</style>  
  
<div>  
  
    <div>  
  
        <p>This is a paragraph inside stylized divs</p>  
  
    </div>  
  
</div>
```

This displays in the web browser as:

This is a paragraph inside stylized divs

Div elements are an important part of building webpages. More advanced usage of div elements will be explored in another course.

Comments

If you want to leave a comment in the code for other developers, it can be added as:

```
<!-- This is a comment -->
```

The comment will not be displayed in the web browser.

Completed

Additional Resources

Learn more Here is a list of resources that may be helpful as you continue your learning journey.

HTML Elements Reference (Mozilla)

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element>

The Form Element (Mozilla)

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form>

What is the Document Object Model? (W3C)

<https://www.w3.org/TR/WD-DOM/introduction.html>

ARIA in HTML (W3C via Github)

<https://w3c.github.io/html-aria/>

ARIA Authoring Practices (W3C)

<https://www.w3.org/TR/wai-aria-practices-1.2/>

Completed

Different types of selectors

When styling a web page, there are many types of selectors available that allow developers to be as broad or as specific as they need to be when selecting HTML elements to apply CSS rules to.

Here you will learn about some of the common CSS selectors that you will use as a developer.

Element Selectors

The element selector allows developers to select HTML elements based on their element type.

For example, if you use `p` as the selector, the rule will apply to all `p` elements on the webpage.

HTML

1

2

```
<p>Once upon a time...</p>
```

```
<p>In a hidden land...</p>
```

CSS

1

2

3

```
p {  
    color: blue;  
}
```

ID Selectors

The ID selector uses the id attribute of an HTML element. Since the id is unique within a webpage, it allows the developer to select a specific element for styling. ID selectors are prefixed with a # character.

HTML

1

```
<span id="latest">New!</span>
```

CSS

1

2

3

```
#latest {  
    background-color: purple;  
}
```

Class Selectors

Elements can also be selected based on the class attribute applied to them. The CSS rule has been applied to all elements with the specified class name. The class selector is prefixed with a . character.

In the following example, the CSS rule applies to both elements as they have the `navigation` CSS class applied to them.

HTML

1

2

```
<a class="navigation">Go Back</a>  
<p class="navigation">Go Forward</p>
```

CSS

1

2

3

```
.navigation {  
    margin: 2px;  
}
```

Element with Class Selector

A more specific method for selecting HTML elements is by first selecting the HTML element, then selecting the CSS class or ID.

The example below selects all `p` elements that have the CSS class `introduction` applied to them.

HTML

1

```
<p class="introduction"></a>
```

CSS

1

2

3

```
p.introduction {  
    margin: 2px;  
}
```

Descendant Selectors

Descendant selectors are useful if you need to select HTML elements that are contained within another selector.

Let's explore an example.

You have the following HTML structure and CSS rule.

HTML

1

2

3

4

5

6

7

8

9

10

11

```
<div id="blog">

<h1>Latest News</h1>

<div>

<h1>Today's Weather</h1>

<p>The weather will be sunny</p>

</div>

<p>Subscribe for more news</p>

</div>

<div>

<h1>Archives</h1>

</div>
```

CSS

1

2

```
#blog h1 {  
  
    color: blue;  
  
}
```

The CSS rule will select all `h1` elements that are contained within the element with the ID `blog`. The CSS rule will not apply to the `h1` element containing the text `Archives`.

The structure of a descendant selector is a CSS selector, followed by a single space character, followed by another CSS selector.

Multiple descendants can also be selected. For example, to select all `h1` elements that are descendants of `div` elements which are descendants of the `blog` element, the selector is specified as follows.

CSS

1

2

3

```
#blog div h1 {  
  
    color: blue;  
  
}
```

Child Selectors

Child selectors are more specific than descendant selectors. They only select elements that are immediate descendants (children) of a selector (the parent).

For example, you have the following HTML structure:

HTML

1

2

3

4

5

6

7

8

```
<div id="blog">

<h1>Latest News</h1>

<div>

<h1>Today's Weather</h1>

<p>The weather will be sunny</p>

</div>

<p>Subscribe for more news</p>

</div>
```

If you wanted to style the `h1` element containing the text `Latest News`, you can use the following child selector:

CSS

1

2

3

```
#blog > h1 {
    color: blue;
}
```

This will select the element with the ID `blog` (the parent), then it will select all `h1` elements that are contained directly in that element (the children). The structure of the child selector is a CSS selector followed by the child combinator symbol `>` followed by another CSS selector.

Note that this will not go beyond a single depth level. Therefore, the CSS rule will **not** be applied to the `h1` element containing the text `Today's Weather`.

:hover Pseudo-Class

A special keyword called a pseudo-class allows developers to select elements based on their state. Don't worry too much about what that means right now. For now, let's look at how the hover pseudo-class allows you to style an element when the mouse cursor hovers over the element. The simplest example of this is changing the color of a hyperlink when it is hovered over. To do this, you add the `:hover` pseudo-class to the end of the selector. In the following example, adding `:hover` to the `a` element will change the color of the hyperlink to orange when it is hovered over.

CSS

```
1  
2  
3  
  
a:hover {  
    color: orange;  
}
```

This pseudo-class is very useful for creating visual effects based on user interaction.

Other Selectors

There are many other CSS selectors available to style your webpage.

Completed

Text and color in CSS

As you design websites, you'll be working a lot with colors and text. There are many different ways to display text and equally as many ways to define colors.

This reading covers how text and color work in CSS.

Color

Colors are used in many CSS properties, for example:

1

2

3

```
p {  
    color: blue;  
}
```

From CSS Version 3, there are five main ways to reference a color.

- By RGB value,
- By RGBA value,
- By HSL value,
- By hex value and
- By predefined color names.

RGB value

RGB is a color model that adds the colors red (R), green (G) and blue (B) together to create colors. This is based on how the human eye sees colors.

Each value is defined as a number between 0 and 255, representing the intensity of that color.

For example, the color red would have the RGB value of 255, 0, 0 since the intensity of the red color would be 100% while blue and green would be 0%.

The color black then would be 0, 0, 0 and the color white 255, 255, 255.

When using RGB values in CSS, they can be defined using the `rgb` keyword:

1

2

3

```
p {  
    color: rgb(255, 0, 0);  
}
```

RGBA value

RGBA is an extension of RGB that add an alpha (A) channel. The alpha channel represents the opacity, or transparency, of the color.

Similar to RGB, this is specified in CSS using the `rgba` keyword:

1

2

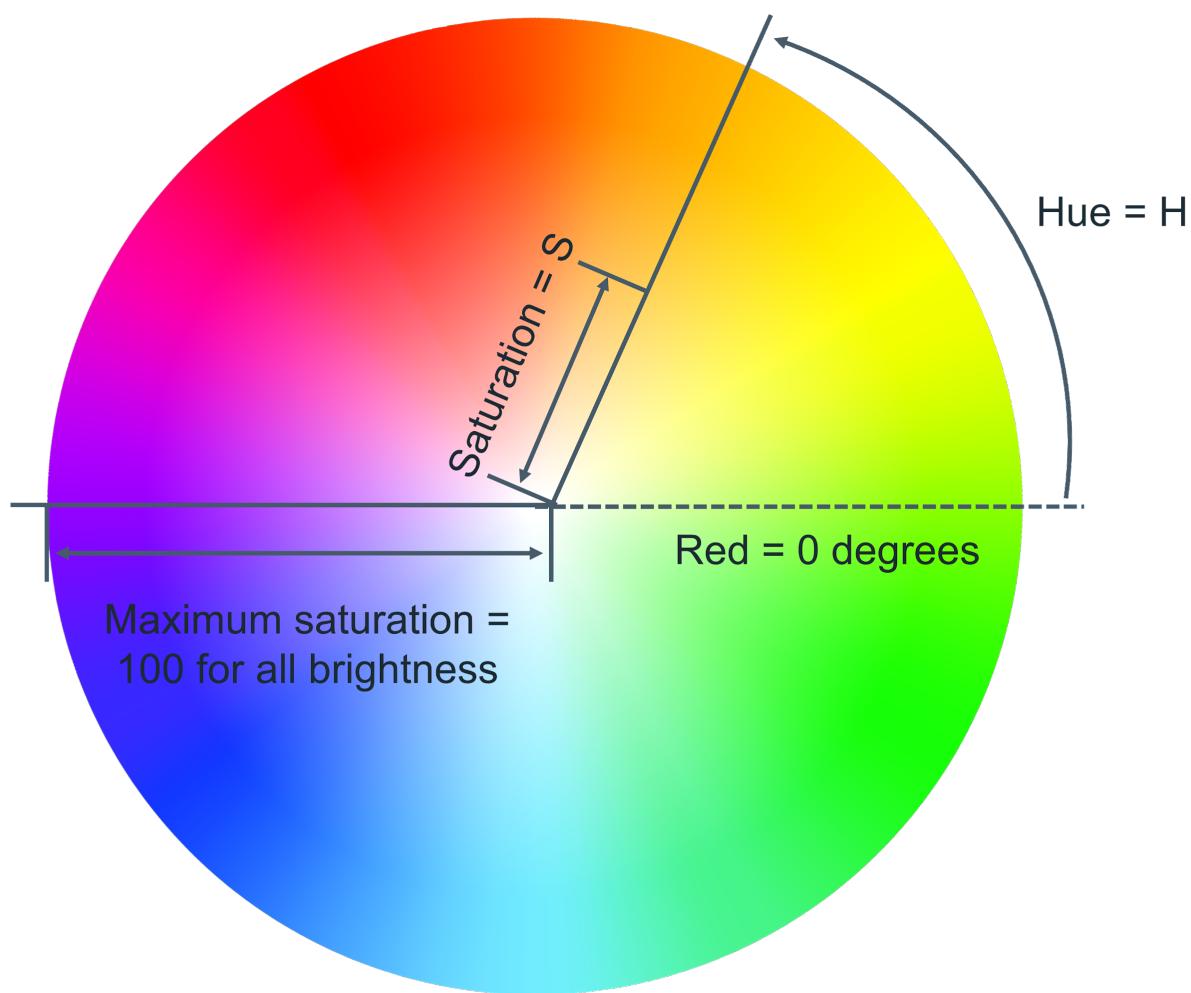
3

```
p {  
color: rgba(255, 0, 0, 128);  
}
```

HSL value

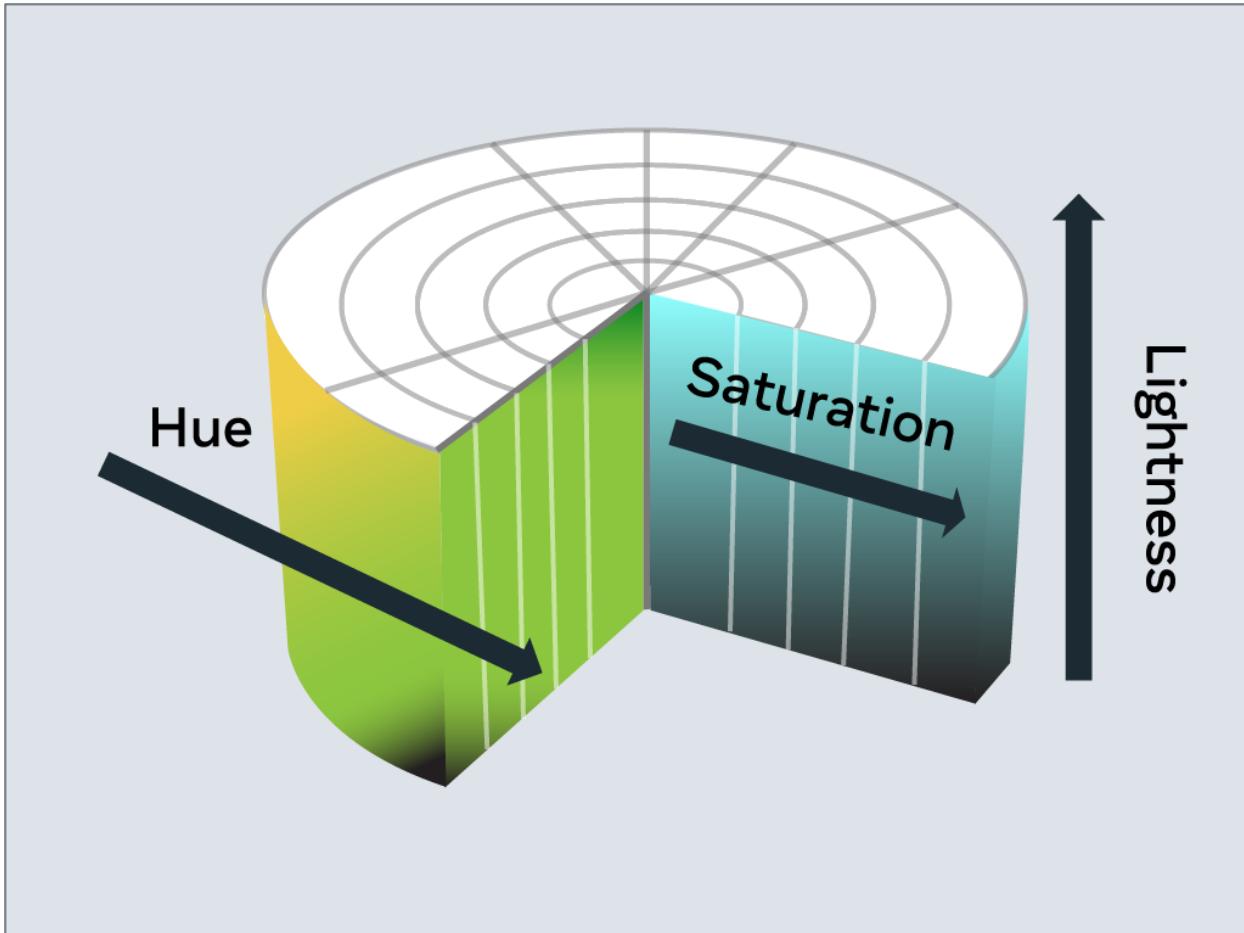
HSL is a newer color model defined as Hue (H), Saturation (S) and Lightness (L). The aim of the model is to simplify mental visualization of the color that the value represents.

Think of a rainbow that has been turned into a full circle. This represents the Hue. The Hue value is the degree value on this circle, from 0 degrees to 360 degrees. 0 is red, 120 is green and 240 is blue.



Saturation is the distance from the center of the circle to its edge. The saturation value is represented by a percentage from 0% to 100% where 0% is the center of the circle and 100% is its edge. For example, 0% will mean that the color is more grey and 100% represents the full color.

Lightness is the third element of this color model. Think of it as turning the circle into a 3D cylinder where the bottom of the cylinder is more black and toward the top is more white. Therefore, lightness is the distance from the bottom of the cylinder to the top. Again, lightness is represented by a percentage from 0% to 100% where 0% is the bottom of the cylinder and 100% is its top. In other words, 0% will mean that the color is more black and 100% is white.



In CSS, you use the `hsl` keyword to define a color with HSL.

1

2

3

```
p {  
    color: hsl(0, 100%, 50%);  
}
```

Hex value

Colors can be specified using a hexadecimal value. If you're unfamiliar with hexadecimal, think of it as a different number set.

Decimal is what you use every day. Digits range from 0 to 9 before tens and hundreds are used.

Hexadecimal is similar, except it has 16 digits. This is counted as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

In fact, you can convert between decimal and hexadecimal. Decimal 10 is equal to hexadecimal A. Hexadecimal F is equal to decimal 15.

Hexadecimal can also go to tens and hundreds. For example, decimal 16 is equal to hexadecimal 10, with 10 being the next number after F.

It can be a little confusing at first but don't worry, there are plenty of converters available if you get stuck.

Colors specified using hexadecimal are prefixed with a # symbol followed by the RGB value in hexadecimal format.

For example, the color red which is RGB 255, 0, 0 would be written as hexadecimal #FF0000.

Again don't worry if you get stuck, there are plenty of converters available for this too!

Predefined color names

Modern web browsers support 140 predefined color names. These color names are for convenience purposes and can be mapped to equivalent hex/RGB/HSL values.

Some common color names available are listed below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

black

silver

gray

white

maroon

red

purple

fuchsia

green

lime

olive

yellow

navy

blue

teal

aqua

Text

With CSS there are many ways to change how text is displayed. In this section, you'll learn the most common text manipulation CSS properties.

Text Color

The `color` property sets the color of text. The following CSS sets the text color for all paragraph elements to red.

```
1
2
3
p {
  color: red;
}
```

Text Font and Size

There are many different fonts to display text on your computer. In simple terms, a font is a collection of text characters written in a specific style and size.

If you've used a word processor before, you're probably familiar with the fonts Times New Roman and Calibri.

To set the font used by text in CSS you use the `font-family` property.

```
1
2
3
p {
  font-family: "Courier New", monospace;
}
```

Since computers vary in what fonts they have installed, it is recommended to include several fonts when using the `font-family` property. These are specified in a fallback order, meaning that if the first font is not available, it will check for the second font. If the second font is not available, then it will check for the third font and so on. If none of the fonts are available, it will use the browser's default font.

To set the size of the font, the `font-size` property is used.

1

2

3

4

```
p {  
    font-family: "Courier New", monospace;  
    font-size: 12px;  
}
```

Text Transformation

Text transformation is useful if you want to ensure the correct capitalization of the text content. In the example below, the CSS rule will change all text in paragraph elements to uppercase using the `text-transform` property:

1

2

3

```
p {  
    text-transform: uppercase;  
}
```

The most commonly used values for the `text-transform` property are: `uppercase`, `lowercase`, `capitalize` and `none`. The default value used is `none`, which means the text displays as it was written in the HTML document.

Text Decoration

The `text-decoration` property is useful to apply additional decoration to text such as underlining and line-through (strikethrough).

1

2

3

```
p {  
    text-decoration: underline;  
}
```

It is possible to set the color, thickness and styling of the decoration too. In the example below, the underline will be a solid red line that is 5 pixels thick.

```
1  
2  
3  
  
p {  
    text-decoration: underline red solid 5px;  
}
```

If this is confusing, don't worry. These properties can be individually set using the `text-decoration-line`, `text-decoration-color`, `text-decoration-style` and `text-decoration-thickness` properties. Let's use the same example again and define it using the individual properties:

```
1  
2  
3  
4  
5  
6  
  
p {  
    text-decoration-line: underline;  
    text-decoration-color: red;  
}
```

```
text-decoration-style: solid;  
  
text-decoration-thickness: 5px;  
  
}
```

The most common `text-decoration-line` values used are: `underline`, `overline`, `line-through` and `none`. `None` is the default value to use no text decoration.

There are many styles available for the `text-decoration-style` property; `solid`, `double`, `dotted`, `dashed` and `wavy`. The `text-decoration-style` property requires the decoration line to be defined. If the decoration style is not specified, `solid` will be used.

Completed

Alignment basics

Let's explore how to align text and HTML elements using CSS.

Let's first focus on horizontal alignment. Vertical alignment is more difficult so you'll explore that later on.

Text Alignment

Aligning text within an HTML element is very simple. To do this, you use the `text-align` CSS property. In the following example, the CSS rule is setting the text of all paragraph elements to be center aligned.

```
1  
  
2  
  
3
```

```
p {  
  
    text-align: center;  
  
}
```

Text alignment can be set to `left`, `right`, `center` and `justify`.

The `justify` alignment spreads the text out so that every line of the text has the same width.

The default alignment is `left` for languages that are left-to-right such as English. For right-to-left languages such as Arabic, the default alignment is `right`.

HTML Element Alignment

HTML element alignment is more complicated than text alignment. To align HTML elements, you must consider the box model and document flow from previous lessons. Aligning an HTML element is done by changing the properties of its box model and how it impacts the document flow.

HTML Element Center Alignment

To center align an element, you set a width on the element and push its margins out to fill the remaining available space of the parent element as in the following HTML structure:

```
1  
2  
3  
4  
  
<div class="parent">  
  
  <div class="child">  
  
    </div>  
  
</div>
```

In your CSS, you'll set the `parent` element to have a red border to visualize the space it occupies:

```
1  
2  
3  
  
.parent {  
  
  border: 4px solid red;  
  
}
```

The `child` element will have a width equal to 50% of the `parent` element with a padding of 20 pixels. Note that `padding: 20px` is shorthand for setting the padding top, bottom, left and right to `20px`. To visualize the space it occupies, set the border to green:

```
1  
2
```

3

4

5

```
.child {  
  width: 50%;  
  padding: 20px;  
  border: 4px solid green;  
}
```

To align the element to the center, set its `margin` property to `auto`. The `auto` will tell the browser to calculate the margin automatically based on the space available.

1

2

3

4

5

6

```
.child {  
  width: 50%;  
  padding: 20px;  
  border: 4px solid green;  
  margin: auto;  
}
```

The result is the `child` element is centered within the `parent` element:



It is important to note that this works because the `div` element is a block-level element.

If you want to align an inline element like `img`, you will need to change it to a block-level element. Similar to the `div` example, you add the `img` to a parent element:

1

2

3

```
<div class="parent">  
  
  
  
</div>
```

The CSS rule then changes the `img` element to a block-level element and sets its margin to `auto`:

1

2

3

4

5

```
.child {  
  
    display: block;  
  
    width: 50%;  
  
    margin: auto;  
  
}
```

To be more precise, in CSS you can set only the left and right margins to auto. This allows you to set the top and bottom margins to specific values if needed.

```
1  
2  
3  
4  
5  
6  
  
.child {  
  
    display: block;  
  
    width: 50%;  
  
    margin-left: auto;  
  
    margin-right: auto;  
  
}
```

HTML Element Left / Right Alignment

The two most common ways to left and right align elements are to use the `float` property and the `position` property.

The `position` property has several value options that impact how the element displays in the document flow. You'll explore how to use the `position` property later on. For now, let's focus on the `float` property.

The `float` property sets an element's position relative to the text content within a parent element. Text will wrap around the element.

In the following example, the image will be aligned to the right of the `div` element. The text content will wrap around the image:

HTML

```
1  
2  
3
```

```
<div class="parent">

     Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Curabitur eu odio eget leo auctor porta sit
amet sit amet justo. Donec fermentum quam in diam volutpat, at lacinia
diam placerat. Aenean quis feugiat sem. Suspendisse a dui massa. Phasellus
scelerisque, mi vestibulum iaculis tristique, orci tellus gravida nisi, in
pellentesque elit massa ut lorem. Sed elementum ornare nunc vel cursus.
Duis sed enim in nulla efficitur convallis sed eget dolor. Curabitur
scelerisque eros erat, in vulputate dolor consequat vel. Praesent ac
sapien condimentum, ultricies libero at, auctor orci. Curabitur ut augue
ac massa convallis faucibus sed in magna. Phasellus scelerisque auctor est
a auctor. Nam laoreet sem sapien, porta imperdiet urna laoreet eu. Morbi
dolor turpis, congue id bibendum eget, viverra et risus. Quisque vitae
erat id tortor ullamcorper maximus.
```

```
</div>
```

CSS

1

2

3

```
.child {

    float: right;

}
```

The following displays in the web browser:

Curabitur eu odio eget leo auctor porta sit amet sit amet justo. Donec fermentum quam in diam volutpat, at lacinia diam placerat. Aenean quis feugiat sem. Suspendisse a dui massa. Phasellus scelerisque, mi vestibulum iaculis tristique, orci tellus gravida nisi, in pellentesque elit massa ut lorem. Sed elementum ornare nunc vel cursus. Duis sed enim in nulla efficitur convallis sed eget dolor. Curabitur scelerisque eros erat, in vulputate dolor consequat vel. Praesent ac sapien condimentum, ultricies libero at, auctor orci. Curabitur ut augue ac massa convallis faucibus sed in magna. Phasellus scelerisque auctor est a auctor. Nam laoreet sem sapien, porta imperdiet urna laoreet eu. Morbi dolor turpis, congue id bibendum eget, viverra et risus. Quisque vitae erat id tortor ullamcorper maximus.

A simple blue line-art icon representing a user profile. It consists of a circle containing a stylized 'U' shape, which represents a person's head and shoulders.



Completed

Additional resources

Learn more Here is a list of resources that may be helpful as you continue your learning journey.

CSS Reference (Mozilla)

<https://developer.mozilla.org/en-US/docs/Web/CSS/Reference>

[HTML and CSS: Design and build websites by Jon Duckett](#)

<https://www.amazon.com/HTML-CSS-Design-Build-Websites/dp/1118008189/>

CSS Definitive Guide by Eric Meyer

<https://www.amazon.com/CSS-Definitive-Guide-Visual-Presentation/dp/1449393195/>

Completed

Bootstrap

Bootstrap is often described as a way to "build fast, responsive sites" and it is a "feature-packed, powerful, and extensible frontend toolkit".

Some people refer to it as a "front-end" framework, and some are trying to be more specific by referring to it as a "CSS framework" or a "CSS library".

So, what is Bootstrap?

Simply put, Bootstrap is a library of CSS and JavaScript code that you can combine to quickly build visually appealing websites.

Modern web development is all about **components**. Small pieces of reusable code that allow you to build websites quickly. Bootstrap comes with multiple components for very fast construction of multiple components, or parts of components.

Another important aspect of modern development is **responsive grids** which allow web pages to adapt their layout and content depending on the device in which they are viewed. Bootstrap comes with a pre-made set of CSS rules for building a responsive grid.

Bootstrap is very popular amongst developers as it saves development time and provides a way for developers to build visually appealing prototypes and websites.

Bootstrap saves significant time because all the CSS code that styles its grid and pre-built components is already written. Instead of having to have a high level of expertise in various CSS concepts, you can just use the existing Bootstrap CSS classes to produce nicely-looking websites. This is indispensable when you need to quickly iterate on website layouts.

Once you know how Bootstrap works, you'll have enough knowledge to tweak its styling and a whole new world of development opens up to you.

Since Bootstrap is so popular, understanding how to work with it is a prerequisite in many web development companies. Additionally, you can be safe in knowing that both you and your team members have a common design system and you don't have to spend time deciding how to build one. You are free to jump from team to team, from project to project, even from one company to another, and you don't need to re-learn "their way of doing things".

All of these points make investing time to learn Bootstrap a great way to boost your web development skills. In this lesson, you'll be introduced to the core concepts of Bootstrap and learn how to build web pages using it.

Completed

Using Bootstrap documentation

Bootstrap comes with detailed documentation on setting up and using the features available in its library. The documentation is clear and has many code examples to help you get started.

In this reading, you'll explore the frequently used documentation sections.

The documentation for Bootstrap is currently available at the following link.

<https://getbootstrap.com/docs>

Navigating the documentation

The sidebar on the webpage allows you to navigate through the different sections of the documentation. There is also a search box if you need to search for a specific piece of information.

[Ctrl + /](#)[› Getting started](#)[› Customize](#)[› Layout](#)[› Content](#)[› Forms](#)[› Components](#)[› Helpers](#)[› Utilities](#)[› Extend](#)[› About](#)

Introduction

Get started with Bootstrap, t
responsive, mobile-first sites.

Quick start

Looking to quickly add Bootstrap to you
or need to download the source files? [H](#)

CSS

Copy-paste the stylesheet [link](#) into yo

Layout

The layout section of the documentation describes how to use the grid system of Bootstrap. This covers what you've learned so far and includes more advanced usage such as offsets, column alignment, auto-layout and variable width columns.

> Getting started

> Customize

✓ Layout

Breakpoints

Containers

Grid

Columns

Gutters

Utilities

Z-index

> Content

> Forms

> Components

> Helpers

> Utilities

> Extend

> About

Migration

Example

Bootstrap's grid system uses a series of containers, rows, and columns to layout and align content. It's built with [flexbox](#) and is fully responsive. Below is an example and an in-depth explanation for how the grid system comes together.

New to or unfamiliar with flexbox? [Read this CSS Tricks flexbox guide](#) for background, terminology, guidelines, and code snippets.

Column	Column	Column
--------	--------	--------

Copy

```
<div class="container">
  <div class="row">
    <div class="col">
      Column
    </div>
    <div class="col">
      Column
    </div>
    <div class="col">
      Column
    </div>
  </div>
</div>
```

On this page

Example

How it works

Grid options

Auto-layout columns

Equal-width

Setting one column width

Variable width content

Responsive classes

All breakpoints

Stacked to horizontal

Mix and match

Row columns

Nesting

Sass

Variables

Mixins

Example usage

Customizing the grid

Columns and gutters

Grid tiers

Content

The content section of the documentation describes Bootstrap's default text styling and how to use responsive images and tables. You've learned the basics of these earlier on and this section goes into further detail.

> Getting started

> Customize

✓ Content

Reboot

Typography

Images

Tables

Figures

> Forms

> Components

> Helpers

> Utilities

> Extend

> About

Tables

[View on GitHub](#)

Documentation and examples for opt-in styling of tables (given their prevalent use in JavaScript plugins) with Bootstrap.

Overview

Due to the widespread use of `<table>` elements across third-party widgets like calendars and date pickers, Bootstrap's tables are **opt-in**. Add the base class `.table` to any `<table>`, then extend with our optional modifier classes or custom styles. All table styles are not inherited in Bootstrap, meaning any nested tables can be styled independent from the parent.

Using the most basic table markup, here's how `.table`-based tables look in Bootstrap.

#	First	Last	Handle
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat

On this page

Overview

Variants

Accented tables

Striped rows

Hoverable rows

Active tables

How do the variants and accented tables work?

Table borders

Bordered tables

Tables without borders

Small tables

Vertical alignment

Nesting

How nesting works

Anatomy

Table head

Table foot

Captions

Responsive tables

Forms

The forms section of the documentation describes how to build forms using Bootstrap's styles. The library has many CSS rules to improve your form's user interface and experience. Below are some features you'll frequently use as a developer:

> Getting started

> Customize

> Layout

> Content

Forms

Overview

Form control

Select

Checks & radios

Range

Input group

Floating labels

Layout

Validation

Forms

[View on GitHub](#)

On this page

Overview

Form text

Disabled forms

Accessibility

Sass

Variables

Form control

Style textual inputs and textareas with support for multiple states.

Checks & radios

Use our custom radio buttons and checkboxes in forms for selecting input options.

Input group

Attach labels and buttons to your inputs for increased semantic value.

Select

Improve browser default select elements with a custom initial appearance.

Range

Replace browser default range inputs with our custom version.

Floating labels

Create beautifully simple form labels that float over your input fields.

Form Styling

Bootstrap includes CSS rules to improve the visual style of input elements.

For example:

Your Email Address

jane@email.com

This table outlines the different HTML form elements and which Bootstrap CSS class should be used for them.

Form Element

CSS class

`input`

`form-control`

`input type="checkbox"`

`form-check-input`

`input type="radio"`

`form-check-input`

`input type="range"`

`form-range`

`select`

`form-select`

Using these CSS classes will style the elements appropriately for different input types, sizings and states. More information is available on the [Forms documentation page](#).

Switches

If you've used an app on your mobile device, you're probably familiar with the switch input type.



New Feature

Bootstrap includes CSS rules to style checkbox input elements as switches.

To do this:

1. Add the `input` to a `div` element.
2. On the `div` element, apply the `form-check` and `form-switch` CSS classes.
3. On the `input` element, add the `form-check-input` CSS class.

1

2

3

```
<div class="form-check form-switch">  
  <input class="form-check-input" type="checkbox">  
</div>
```

More information is available in the [Switches section of the documentation](#).

Input Groups

Input groups are useful for providing additional content to the input field. For example, if you wanted to request the user to input a US dollar amount, you can use an input group to show the dollar symbol and cents amount.



To do this:

1. Add the `input` to a `div` element.
2. Apply the `input-group` CSS classes on the `div` element.
3. Add a `span` element before and/or after the `input` element and apply the `input-group-text` CSS class to it. The text content is then added inside the `span` element.

1

2

3

4

5

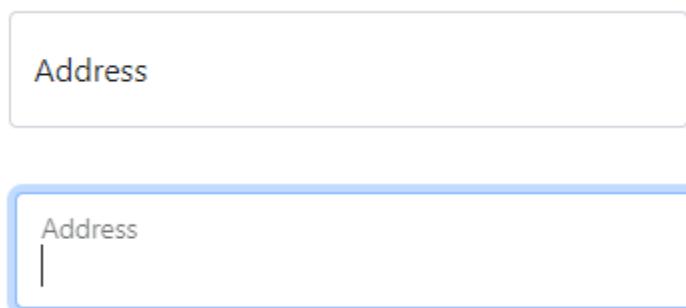
```
<div class="input-group">  
  <span class="input-group-text">$</span>
```

```
<input type="text" class="form-control">  
  
<span class="input-group-text">.00</span>  
  
</div>
```

More information is available on the [Input Groups documentation page](#).

Floating Labels

Floating labels help provide form information to the user as part of the input itself. These are different from regular form placeholders. The information stays visible if the user is interacting with the element or if the element has content.



To do this, add the `input` to a `div` element. On the `div` element, apply the `form-floating` CSS classes.

- 1
- 2
- 3
- 4

```
<div class="form-floating">  
  
  <input type="email" class="form-control" id="addressInput"  
  placeholder="Address">  
  
  <label for="addressInput">Address</label>  
  
</div>
```

More information is available on the [Floating Labels documentation page](#)

Components

As you have learned, Bootstrap comes with many pre-made UI elements and styles to help speed up your development.

Some of these components require Javascript to work, while others only require CSS classes applied to HTML elements. The Components section of the documentation explains these requirements on each component page and provides many code examples.

> Getting started
> Customize
> Layout
> Content
> Forms
v Components
 Accordion
 Alerts
 Badge
 Breadcrumb
 Buttons
 Button group
 Card
 Carousel

Buttons

[View on GitHub](#)

Use Bootstrap's custom button styles for actions in forms, dialogs, and more with support for multiple sizes, states, and more.

Examples

Bootstrap includes several predefined button styles, each serving its own semantic purpose, with a few extras thrown in for more control.



On this page

Examples
Disable text wrapping
Button tags
Outline buttons
Sizes
Disabled state
Block buttons
Button plugin
Toggle states
Methods
Sass
Variables
Mixins
Loops

Conclusion

Now that you are familiar with how to use the Bootstrap documentation, maybe try some new components and styles on a webpage that you've previously built.

Completed

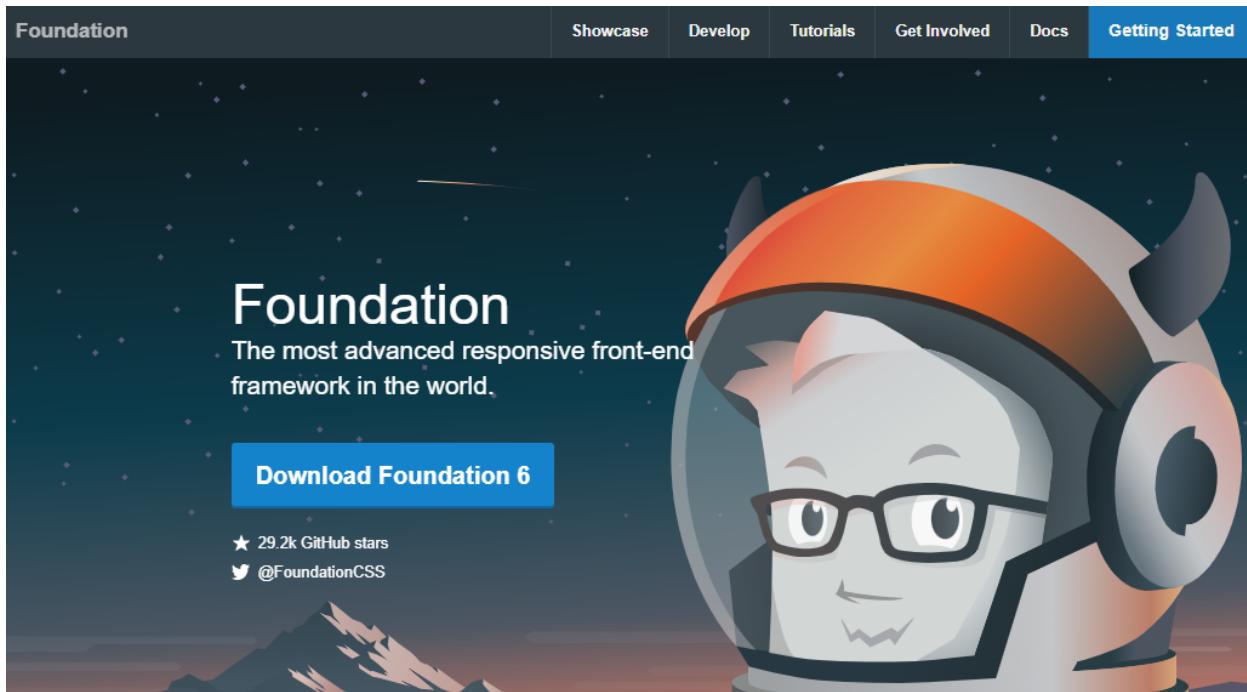
Other CSS frameworks and libraries

As a developer, you'll use many CSS libraries and frameworks throughout your career. As you move on to different projects and as technologies advance, knowing what solutions are available is critical. While Bootstrap is one of the most popular CSS libraries, many others are available, each with different purposes, designs and technical approaches. This reading will introduce you to other popular CSS libraries and frameworks.

Foundation

[Official Website](#)

Foundation is a framework for building user interfaces similar to Bootstrap. It is used by many large companies such as Pixar, Polar and Sonos. One prominent feature of Foundation is that it can be used to style content for sending via email.



Pure.css

Official Website

Pure.css is another library for building user interfaces. While it doesn't have as many features as Bootstrap, it is designed to be minimal in file size. Smaller file sizes improve loading times for web pages as there is less data to transfer from the web server. If your next project is focused on minimal loading time, this library is worth considering.

The screenshot shows the Pure.css website's landing page. On the left, there is a vertical navigation menu with a black background and white text, listing categories such as "PURE", "Get Started", "Layouts", "Base", "Grids", "Forms", "Buttons", "Tables", "Menus", "Tools", and "Customize". To the right of the menu, the Pure.css logo is displayed, consisting of a blue square with a white letter "P" and the text "Pure.CSS" in a large, sans-serif font. Below the logo, a descriptive text reads "A set of small, responsive CSS modules that you can use in every web project." At the bottom of the page, there is a code snippet showing a link tag for the CSS file, followed by two buttons: "Get Started" and "View on GitHub".

Tailwind CSS

[Official Website](#)

Tailwind CSS is a CSS framework that uses a utility-based approach for its CSS rules. This means that the framework provides many CSS classes with a single purpose. For example, the CSS class `pt-6` sets the padding-top CSS property to 6 pixels. This means that you can be precise in applying styling to your HTML without writing CSS. The advantage to this is that it is more flexible for customizing your webpage's design using the framework. However, the disadvantage is that if multiple developers are working on a project, it could lead to inconsistent design if the team is not strict on its design rules.

The screenshot shows the official Tailwind CSS website. At the top, there is a navigation bar with the Tailwind logo, "tailwindcss" text, and links for "Docs", "Components", and "Blog". To the right of these are icons for dark mode and refresh. Below the navigation is a large, bold headline: "Rapidly build modern websites without ever leaving your HTML.". Underneath the headline is a descriptive text: "A utility-first CSS framework packed with classes like `flex`, `pt-4`, `text-center` and `rotate-90` that can be composed to build any design, directly in your markup.". At the bottom of the main section are three buttons: a dark "Get started" button, a search bar with placeholder "Quick search..." and keyboard shortcut "Ctrl K", and a light-colored "Install" button.

UIKit

[Official Website](#)

UIKit is a lightweight CSS framework featuring a small set of responsive components. Its simple design allows developers to easily customize the style rules and visuals.



Ulkit

PRO

DOCUMENTATION

CHANGELOG

DOWNLOAD



A lightweight and modular front-end framework
for developing fast and powerful web interfaces.

[GET STARTED](#)

[GITHUB](#)

MVP.css

[Official Website](#)

MVP.css is a small CSS library that automatically styles HTML elements without needing to apply CSS classes to them. The library aims to allow a developer to quickly prototype a user interface without worrying about the final design, while still being visually appealing. MVP comes from the technical term Minimal Viable Product, a product with sufficient features to demo to customers or other business stakeholders.



[Home](#) [Docs](#) [FAQ](#) [Dark Mode](#) [GitHub ↗](#)

A minimalist stylesheet for HTML elements

No class names, no frameworks, just *semantic* HTML and you're done.

[Download HTML ↗](#)

[Download MVP.css \(8kb\) ↗](#)

PRO TIP Add this code to a new HTML file:

```
<link rel="stylesheet" href="https://unpkg.com/mvp.css">
```

Conclusion

If you're curious to learn more about these frameworks, their websites feature set up guides, tutorials and documentation to get started. It is a good exercise to compare and contrast different libraries and frameworks to understand different workflows available to you as a developer.

Completed

Additional Resources

Bootstrap Official Website

<https://getbootstrap.com/>

Bootstrap 5 Foundations by Daniel Foreman

<https://www.amazon.com/Bootstrap-Foundations-Mr-Daniel-Foreman/dp/B0948GRS8W/>

Responsive Web Design with HTML5 and CSS by Ben Frain

<https://www.amazon.com/Responsive-Web-Design-HTML5-CSS/dp/1839211563/>

Bootstrap Themes

<https://themes.getbootstrap.com/>

Completed

Case Study: Why did Facebook engineers create React?

There are a lot of JavaScript Model-View-Controller (MVC) frameworks out there. Why did we build React and why would you want to use it?

React isn't an MVC framework.

React is a library for building composable user interfaces. It encourages the creation of reusable UI components which present data that changes over time.

React doesn't use templates.

Traditionally, web application UIs are built using templates or HTML directives. These templates dictate the full set of abstractions that you are allowed to use to build your UI.

React approaches building user interfaces differently by breaking them into **components**. This means React uses a real, full-featured programming language to render views, which we see as an advantage over templates for a few reasons:

- **JavaScript is a flexible, powerful programming language** with the ability to build abstractions. This is incredibly important in large applications.
- By unifying your markup with its corresponding view logic, React can actually make views **easier to extend and maintain**.
- By baking an understanding of markup and content into JavaScript, there's **no manual string concatenation** and therefore less surface area for XSS vulnerabilities.

We've also created [JSX](#), an optional syntax extension, in case you prefer the readability of HTML to raw JavaScript.

React updates are dead simple.

React really shines when your data changes over time.

In a traditional JavaScript application, you need to look at what data changed and imperatively make changes to the DOM to keep it up-to-date. Even AngularJS, which provides a declarative interface via directives and data binding [requires a linking function to manually update DOM nodes](#).

React takes a different approach.

When your component is first initialized, the `render` method is called, generating a lightweight representation of your view. From that representation, a string of markup is produced and injected into the document. When your data changes, the `render` method is called again. In order to perform updates as efficiently as possible, we diff the return value from the previous call to `render` with the new one and generate a minimal set of changes to be applied to the DOM.

The data returned from `render` is neither a string nor a DOM node — it's a lightweight description of what the DOM should look like.

We call this process **reconciliation**. Check out [this jsFiddle](#) to see an example of reconciliation in action.

Because this re-render is so fast (around 1ms for TodoMVC), the developer doesn't need to explicitly specify data bindings. We've found this approach makes it easier to build apps.

HTML is just the beginning.

Because React has its own lightweight representation of the document, we can do some pretty cool things with it:

- Facebook has dynamic charts that render to `<canvas>` instead of HTML.
- Instagram is a “single page” web app built entirely with React and `Backbone.Router`. Designers regularly contribute React code with JSX.
- We’ve built internal prototypes that run React apps in a web worker and use React to drive **native iOS views** via an Objective-C bridge.
- You can run React on the server for SEO, performance, code sharing and overall flexibility.
- Events behave in a consistent, standards-compliant way in all browsers (including IE8) and automatically use event delegation.

Head on over to <https://reactjs.org> to check out what we have built.

Completed

The Virtual DOM

React builds a representation of the browser Document Object Model or DOM in memory called the virtual DOM. As components are updated, React checks to see if the component’s HTML code in the virtual DOM matches the browser DOM. If a change is required, the browser DOM is updated. If nothing has changed, then no update is performed.

As you know, this is called the **reconciliation** process and can be broken down into the following steps:

Step 1: The virtual DOM is updated.

Step 2: The virtual DOM is compared to the previous version of the virtual DOM and checks which elements have changed.

Step 3: The changed elements are updated in the browser DOM.

Step 4: The displayed webpage updates to match the browser DOM.

As updating the browser DOM can be a slow operation, this process helps to reduce the number of updates to the browser DOM by only updating when it is necessary.

But even with this process, if a lot of elements are updated by an event, pushing the update to the browser DOM can still be expensive and cause slow performance in the web application.

The React team invested many years of research into solving this problem. The outcome of that research is what’s known as the React Fiber Architecture.

The Fiber Architecture allows React to incrementally render the web page. What this means is that instead of immediately updating the browser DOM with all virtual DOM changes, React can spread the update over time. But what does “over time” mean?

Imagine a really long web page in the web browser. If the user scrolls to the bottom, the top of the web page is no longer visible. The user then clicks a button on the bottom of the web page that updates some text on the top of the web page.

But the top of the page isn't visible. Therefore, why update it immediately?

Perhaps there is text currently displayed on the bottom of the page that also updates when the button is clicked. Wouldn't that be a higher priority to update than the non-visible text?

This is the principle of the React Fiber Architecture. React can optimize when and where updates occur to the browser DOM to significantly improve application performance and responsiveness to user input. Think of it as a priority system. The highest priority changes, the elements visible to the user, are updated first. While lower priority changes, the elements not currently displayed, are updated later.

While you're unlikely to interact with the virtual DOM and Fiber Architecture yourself, it's good to know what's going on if issues occur during the development of your web application.

There are many tools available to help you investigate how React is processing your webpage. The official React Developer Tools web browser plugin developed by Meta will be one of the key tools in your developer toolbox. So, if you do have to look deeper into the code, you'll have the right toolbox available to help you. These tools will be explored later on.

Completed

Alternatives to React

React is a library and not a framework. This means you'll often use other JavaScript libraries with it to build your application. In this reading, you will be briefly introduced to some JavaScript libraries commonly used with React.

Lodash

[Official Website](#)

As a developer, there's a lot of logic you'll commonly write across applications. For example, you might need to sort a list of items or round a number such as `3.14` to `3`. Lodash provides common logic such as these as a utility library to save you time as a developer.



Lodash

A modern JavaScript utility library delivering modularity, performance & extras.

[Documentation](#)

[FP Guide](#)

```
_.defaults({ 'a': 1 }, { 'a': 3, 'b': 2 });
// → { 'a': 1, 'b': 2 }
_.partition([1, 2, 3, 4], n => n % 2);
// → [[1, 3], [2, 4]]
```

Luxon

[Official Website](#)

You'll be working with dates and times often as a developer. Think of viewing a list of orders and when they were placed, or displaying a calendar schedule for an event. Dates and times are everywhere.

Luxon helps you work with dates and times by providing functions to manipulate and display them. For example, think of how dates are formatted in different countries. In the United States the format is **Month Day Year** but in Europe it is **Day Month Year**. This is one area where Luxon can help you display the date in the user's local format.



Luxon

2.x

A powerful, modern, and friendly wrapper for JavaScript dates and times.

DateTimes, Durations, and Intervals
Immutable, chainable, unambiguous API.
Native time zone and Intl support (no locale or tz files)

[GitHub](#) [Get started](#)

Redux

[Official Website](#)

When building a web application, you'll need to keep track of its state. Think of when you shop online. The web application tracks items currently in your shopping cart. When you remove an item from the cart, the application needs to update what displays on the screen. This is where Redux comes in. It helps you manage your application state and even has advanced features such as undo and redo.



Search CTRL K

Redux

A Predictable State Container for JS Apps

[Get Started](#)

Axios

[Official Website](#)

As a developer you'll be communicating with APIs over HTTP frequently. The Axios library helps to simplify sending HTTP requests and processing the response. It also provides advanced features allowing you to cancel requests and to change data received from the web server before your application uses the data.



Get Started

Promise based HTTP client for the browser and node.js

Axios is a simple promise based HTTP client for the browser and node.js. Axios provides a simple to use library in a small package with a very extensible interface.

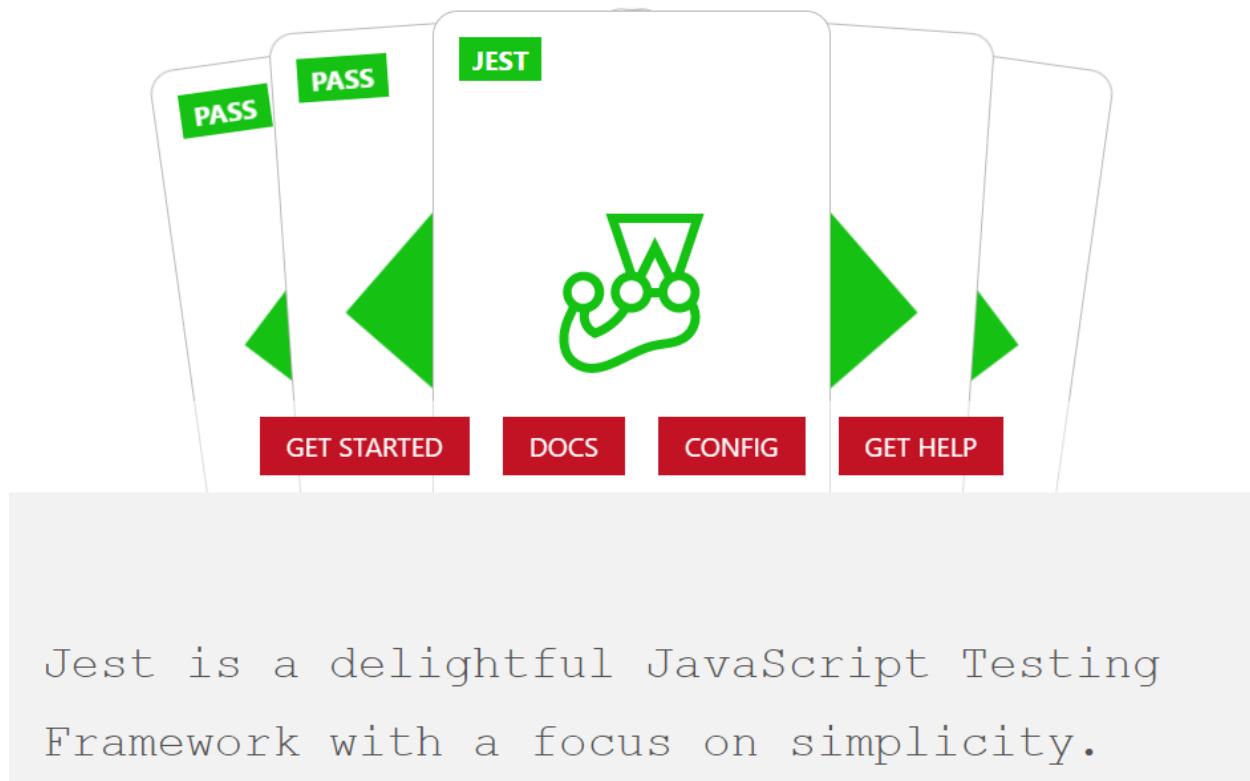
[Get Started](#)

[View on GitHub](#)

Jest

[Official Website](#)

It is good practice to write automated tests for your code as a professional developer. The jest library helps you to do this and works with many libraries and frameworks. It also provides reporting utilities such as providing information on how much of your code is tested by your automated tests.



The image shows the Jest testing framework's landing page. At the top, there are three mobile phone icons. The first phone has a green "PASS" badge at the top and a green chevron pointing right at the bottom. The second phone also has a green "PASS" badge at the top and a green chevron pointing right at the bottom. The third phone has a green "JEST" badge at the top and a green chevron pointing right at the bottom. Below these phones are four red buttons: "GET STARTED", "DOCS", "CONFIG", and "GET HELP". In the center, there is a green icon of a hand holding a trophy. Below the phones, there is a large grey box containing the text: "Jest is a delightful JavaScript Testing Framework with a focus on simplicity.".

Conclusion

If you're curious to learn more about these libraries, their websites feature setup guides, tutorials and documentation to get started. These libraries will be covered later on.

Completed

Additional Resources

Learn more Here is a list of resources that may be helpful as you continue your learning journey.

React Official Website

<https://reactjs.org/>

Choosing between Traditional Web Apps and Single Page Apps (Microsoft)

<https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps>

React Source Code (Github)

<https://github.com/facebook/react>

Introduction to React.js

The original video recorded at Facebook in 2013.

https://youtu.be/XxVg_s8xAms

Completed

About the Ungraded Lab: Improve your Bio page with Bootstrap

In this Ungraded Lab, you will update your biographical page from Week 2 - Introduction to HTML5 and CSS to use Bootstrap.

The expected outcome is a two-column biographical page with your name and a photo in the left column and your favorite music artists and films in the right column.

The image below shows how your page should look once you finish the assessment.



Jane

Favorite Music Artists

- Metallica
- Bob Marley
- Madonna
- The Beatles
- Pink Floyd

Favorite Films

1. Pulp Fiction
2. The Godfather
3. The Lord of the Rings
4. Iron Man
5. Inception

[My Meta Profile](#)

Completed

Exemplar

By updating your biographical page to use Bootstrap, your new page should be similar to the image below.

Jane



Favorite Music Artists

- Metallica
- Bob Marley
- Madonna
- The Beatles
- Pink Floyd

Favorite Films

1. Pulp Fiction
2. The Godfather
3. The Lord of the Rings
4. Iron Man
5. Inception

[My Meta Profile](#)

Your HTML file structure and content should be similar to the snippet below. Note where the Bootstrap CSS classes were used in the different HTML elements.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

```
<!DOCTYPE html>

<html>

<head>

    <title>My Bio Page</title>

    <link href="bootstrap.min.css" rel="stylesheet">

</head>

<body>

    <div class="container">

        <div class="row">

            <div id="bio" class="col-12 col-lg-6 text-center">

                <h1>Jane</h1>

            </div>

            <div id="more" class="col-12 col-lg-6">

                <h2>Favorite Music Artists</h2>

            </div>

        </div>

    </div>

</body>
```

```
<ul>

<li>Metallica</li>

<li>Bob Marley</li>

<li>Madonna</li>

<li>The Beatles</li>

<li>Pink Floyd</li>

</ul>

<h2>Favorite Films</h2>

<ol>

<li>Pulp Fiction</li>

<li>The Godfather</li>

<li>The Lord of the Rings</li>

<li>Iron Man</li>

<li>Inception</li>

</ol>

<a href="My Meta Profile"></a>

</div>

</div>

<script src="bootstrap.bundle.min.js"></script>
```

```
</body>
```

```
</html>
```

As a developer, Bootstrap is one of the many libraries in your toolkit to help build web applications. Bootstrap has many utilities and components. We encourage you to read their documentation and experiment further with styling your webpage.

Completed

Next steps after Introduction to Web Development

Congratulations! You've completed this course and taken another step toward improving your knowledge, skills and qualifications.

In the next course, you'll learn about JavaScript. JavaScript is the programming language that powers the modern web. You will learn the basic concepts of web development with JavaScript. You will work with functions, objects, arrays, variables, data types, the HTML DOM and much more. You will learn how to use JavaScript within the React framework and discover interactive possibilities with modern JavaScript technologies. Finally, you will learn about the practice of testing code and how to write a unit test using Jest.

Completed

Course2-Programming with JavaScript

Course syllabus

This course is the second of a series that aims to help you learn more about web development.

In this course, you will explore the following:

Module 1: Introduction to JavaScript

In this module, you are introduced to JavaScript. You'll learn why JavaScript is so integral to software development. And you'll get an overview of how to write JavaScript code inside the browser.

Furthermore, you will learn about the most common operators as well as conditional statements and loops.

After completing this module, you will be able to:

- Explain the importance of JavaScript in software development
- Demonstrate how to write JavaScript code inside the browser
- Demonstrate how to write basic JavaScript code
- List common operators, conditional statements and loops
- Demonstrate how to use variables and output their value in the console

Module 2: The building blocks of a program

Here you'll learn how to use objects, arrays and functions. In addition, you will learn about the most common built-in methods, and the difference between undefined, null and empty strings. And you'll explore both error handling and defensive programming.

After completing this module, you will be able to:

- Build and use objects, arrays, and functions
- List some common built-in methods on built-in objects
- Describe handling bugs and errors using try, catch, throw, and defensive programming
- Explain the difference between undefined, null, and empty strings
- Demonstrate how to write basic code using arrays, objects and functions

Module 3: Programming paradigms

This module is about functional programming and the object oriented programming paradigm. You will learn what scope is in JavaScript. You'll explore the differences between var, let and const. And you'll learn how to use classes and inheritance in object oriented programming. Additionally, you'll explore how to use write JavaScript using modern features like spread and rest. You will build code that can manipulate the DOM and handle events. And you will use JSON in JavaScript.

After completing this module, you will be able to:

- Outline the tenets of the functional programming and object oriented programming paradigm
- Describe how scope works in JavaScript
- List the differences between var, let, and const
- Use classes and inheritance in OOP in JavaScript

- Write JavaScript code using more modern features like spread, rest, template strings and modules
- Build code that manipulates the DOM and handles events
- Use JSON in JavaScript

Module 4: Testing and compatibility

Here you will learn about Node.js and npm. And you will explore how to install npm packages and how to work with package.json. Furthermore, you will learn about testing in JavaScript and you'll code a simple unit test in Jest.

After completing this module, you will be able to:

- Describe Node.js and npm
- Explain how to install npm packages
- Describe how to work with package.json
- Explain the process of testing in JavaScript
- List the three most prevalent kinds of testing
- Demonstrate how to code a simple unit test in Jest

Module 5: Graded assessment

In the final module, you'll learn about the graded assessment. After you complete the individual units in this module, you'll synthesize the skills you gained from the course to create code for the "Little lemon receipt maker".

You'll also have the opportunity to reflect on the course content and the learning path that lies ahead.

Completed

How to be successful in this course

Taking an online course can be overwhelming. How do you learn at your own pace and successfully achieve your goals?

Here are some general tips that can help you stay focused and on track.

Set daily goals for studying

Ask yourself what you hope to accomplish in your course each day. Setting a clear goal can help you stay motivated and beat procrastination. The goal should be specific and easy to measure, such as "I'll watch all the videos in Module 2 and complete the first programming assignment". And don't forget to reward yourself when you make progress towards your goal!

Create a dedicated study space

It's easier to recall information if you're in the same place where you first learned it, so having a dedicated space at home to take online courses can make your learning more effective. Remove any distractions from the space and if possible, make it separate from your bed or sofa. A clear distinction between where you study and where you take breaks can help you focus.

Schedule time to study on your calendar

Open your calendar and choose a predictable, reliable time that you can dedicate to watching lectures and completing assignments. This helps ensure that your courses won't become the last thing on your to-do list.

Tip: You can add deadlines for a Coursera course to your Google calendar, Apple calendar, or another calendar app.

Keep yourself accountable

Tell your friends about the courses you're taking, post achievements to your social media accounts or blog about your homework assignments. Having a community and support network of friends and family to cheer you on makes a difference!

Actively take notes

Taking notes can promote active thinking, boost comprehension and extend your attention span. It's a good strategy to internalize knowledge whether you're learning online or in the classroom. So, grab a notebook or find a digital app that works best for you and start synthesizing key points.

Tip: While watching a lecture on Coursera, you can click the 'Save Note' button below the video to save a screenshot to your course notes and add your own comments.

Join the discussion

Course discussion forums are a great place to ask questions about assignments, discuss topics, share resources and make friends. Our research shows that learners who participate in the discussion forums are 37% more likely to complete a course. So make a post today!

Do one thing at a time

Multitasking is less productive than focusing on a single task at a time. Researchers from Stanford University found that "People who are regularly bombarded with several streams of electronic information cannot pay attention, recall information or switch from one job to another as well as those who complete one task at a time." Stay focused on one thing at a time. You'll absorb more information and complete assignments with greater productivity and ease than if you were trying to do many things at once.

Take breaks

Resting your brain after learning is critical to high performance. If you find yourself working on a challenging problem without much progress for an hour, take a break. Walking outside, taking a shower or talking with a friend can help you to re-energize and even give you new ideas on how to tackle the project.

Your learning journey starts now!

While preparing for the module quiz or working on achieving your learning goals you're encouraged to:

- Work through each lesson in the learning pathway. Try not to skip any activities or lessons unless you are certain that you already know this information well enough to move ahead.
- Take the opportunity to go back and watch a video or read all the information provided before moving on to the next lesson or module.
- Complete all the knowledge and module quizzes and exercises.
- Read the feedback carefully when answering quizzes, as this will help you to reinforce what you are learning.
- Make use of the practical learning environment provided by the exercises. You can gain substantial reinforcement of your learning through the step-by-step application of your skills.

Completed

How to Position Yourself for a New Career

You are well on your way to becoming a software developer.

You took the most important first step: you started.

While this specialization on Coursera will make you into a well-rounded junior developer, you are basically just getting started.

Here are some proven tips to make the transition to your new career as smooth as possible.

Be persistent

Succeeding in your career efforts is not easy. Luckily, it's not too hard either. Consider this new endeavor of learning to code a part of your everyday life.

Make it as much of a routine as possible. Hopefully, it will work like this:

- You wake up,
- You brush your teeth,
- You run some errands,
- And then you write and learn to code.

Obviously, there are things like your school obligations, or your day job, or other places you need to be and things you need to do.

However, if you don't code regularly - preferably on a daily basis - your progress will be slower. Try to set aside some time to consistently code and learn every day. Persistence is key.

Start building simple apps today

Don't wait until you "learn enough". There's always more to learn, and it's best to get started with any kind of a simple project right now.

Even just taking the code from this specialization and tinkering with it will do wonders for your confidence and the speed at which you acquire new knowledge.

Also, the more you practice, the better you'll retain what you've learned.

Having your own projects that you can showcase to others - no matter how small or straightforward shows a track record and dedication. This is something that your future employer might be impressed with, so start today.

Set up a GitHub account

Since we're on the topic of personal projects, head on over to [GitHub](#) and set up your developer profile right away. It's essential to have an account there since you can keep all your projects in a single location that you can access from any computer.

You can almost think of your GitHub account as an additional brain power. No matter how long ago, whatever you've worked on will remain there, waiting for you to peek into and re-familiarise yourself with.

Pair program

Try to find someone at your level or perhaps slightly more knowledgeable than you and ask them to set up a recurring pair programming session.

This works nicely because having a pair programming partner can speed up your learning. You also have someone to be accountable to.

Start a coding blog

Technical communication is important for developers, and just like anything else, you get better with practice.

Starting a coding blog will work the same as having a GitHub account, with some the additional benefits:

- It shows even more dedication - and this increases your chances of getting hired
- It helps you experiment with different technologies
- Setting up your own website is practical learning in its own right and one more project to add to your CV

Collaborate on open source projects

Even if you are just starting out and are really struggling to get into this field, you can still be a valuable contributor to open-source projects.

There are so many open-source projects that are in demand for all kinds of contributors.

Even contributing to a project by fixing some typos in documentation files is a nice start to getting more involved and putting yourself out there.

Get a certificate

Getting certified is always a good thing. The fact that you're reading this lesson right now confirms that you're on your way to receiving a certificate of completion from Coursera!

Keep a positive attitude

As with anything worth doing, you might sometimes get tired, not understanding how something works, and perhaps even feel like giving up.

Remember to stay consistent.

There are always ups and downs in life, but sometimes it's worth it to think of all the things you've achieved so far and use that as motivation to keep at it.

Never stop learning

There's always more to learn in IT, and that's probably the best thing about it. It's the very thing that makes it fun and provides an opportunity for each developer to get ahead in their career.

Completed

How to uncover job opportunities

Learning how to program in JavaScript helps you prepare for a wide range of job opportunities. This is in part because it expands the possibilities of what you can build as a developer.

JavaScript is one of the most in-demand programming languages, as it is used in nearly all active websites. Its versatility enables developers to use popular libraries, plugins, and frameworks such as React, which improves efficiency and productivity.

With all flexibility that programming in JavaScript brings, it is no surprise that there are different careers that you might want to follow. However, no matter what career path you choose, you will always want to learn other technologies like HTML, CSS, React, Node.js, or Python, so you are more marketable.

Let's cover a few of the most common roles you can get if you know how to program in JavaScript.

Mobile Developer There has been a constant increase in the demand for mobile developers as mobile devices' use for accessing the internet has been on the rise. Mobile developers specialize in building apps for platforms like Google's Android and Apple's iOS. Many developers choose to use React Native, which allows them to build one application using JavaScript that works on both Android and iOS devices. Mobile developers work with UX and UI designers and use React's UI capabilities to implement functionalities that customers will use. Mobile developers also make sure that the front-end and the back-end of the applications work seamlessly. Other skills that mobile developers may have include HTML, CSS, Java, Kotlin, Objective-C, C++, C#, among others.

Front-End Developer As the name implies, front-end developers build the user-facing parts of websites and apps. They work closely with designers to implement visual and interactive elements through coding, using HTML, CSS, and of course, JavaScript. They also use libraries and frameworks like React to save time and make their work more efficient. Front-end developers may also be responsible for making sure the final user has a good user experience and that websites and apps behave as expected and are free of errors and bugs.

Back-End Developer Back-end developers work on the back-end of websites and applications.

They can use JavaScript with Node.js to develop back-end functionalities. Among these functionalities are streaming and chat-based applications, as well as JSON APIs and serverless functions. Back-end developers possess additional skills, including professional working knowledge of Python, APIs, cloud infrastructure, and database.

Full-Stack Developer As you might guess, a full-stack developer works with both the front- and back-end of building websites and apps. So, these professionals combine the skills in these two areas. They can use frameworks like React to work on the front-end and Node.js to work on the back-end. They also apply skills in addition to JavaScript to build websites and apps.

Your professional journey ahead

As you embark on the exciting career as a professional developer, you will realize that you will progressively expand your skills to include a wide range of technologies and programming languages besides JavaScript.

If you want to have an idea of the opportunities available in these areas, you might want to check your favorite job search website or app and look for jobs related to JavaScript. You will learn that there is no shortage of opportunities. And as you start reading more about these roles, you will find that JavaScript is often only one of the competencies that employers are looking for.

But don't worry, as you advance in your studies and in your career, you will further develop your skills to focus on the professional path you want to follow.

Good luck on your professional journey!

Completed

Writing your first Javascript code

In this reading, you'll learn about comments in JavaScript. Additionally, you'll learn about the semi-colon in JavaScript: what it does and why it is used. You will then download a browser if you don't have one installed and run your first piece of JavaScript using the Console.

Comments in JavaScript

I've chosen comments as the starting point for two reasons:

1. Their syntax - the way comments are written is easy to understand.
2. Writing comments can empower you as a developer.

First, I'll explain the syntax, and after that, I'll discuss why being able to write comments is so empowering.

Comments in JavaScript: the syntax

There are two varieties of comments in JavaScript:

1. Single-line comments
2. Multi-line comments

A single-line comment is created when you add two forward-slash characters one after the other, without spaces.

1

```
// this is a comment!
```

Anything that follows a single-line comment in JavaScript is ignored by the browser. This means that, essentially, you can write any kind of text, code, characters, emojis, whatever - and the browser will ignore it. A multi-line comment, as its name says, spans for several lines of code and is created with a forward slash and a star. For example:

```
1  
2  
3  
4  
5  
6  
7  
  
/*  
this  
is  
a  
multi-line  
comment  
*/
```

You can also use the multi-line comment syntax on a single line of code, as follows:

```
1  
  
/* this is a multi-line comment on a single line */
```

Why writing comments is empowering

In this course, it is assumed that you've never written a single line of JavaScript code.

With this assumption in mind, consider the effects of what you've just learned, that is, the effects of learning how to write comments in JavaScript:

1. You can now freely express your ideas about any code that you write.
2. You can add comments to any code that already exists.
3. Those comments can be intended for your future self, or for colleagues on your development team.

So, comments are empowering because they facilitate communication with your future self or with your team members, allowing you to ask questions about the code, mark the code as "to do", or as "to improve", or just simply explain what a given piece of code does.

Additionally, you can even comment out some working code in a JavaScript file - to prevent it from running.

Effectively, comments allow you to "switch off" pieces of JavaScript code.

There can be many reasons for that:

1. Trying to understand how a given piece of code works.
2. Testing different solutions to a coding problem - while not having to delete existing code.
3. Debugging - trying to pin-point why your code is broken or behaving unpredictably.

The semi-colon in JavaScript

In the English language, the fullstop or period - the . character - is used to separate thoughts into sentences.

By clearly separating thoughts with the fullstop, you avoid being misunderstood.

In JavaScript, the semi-colon - the ; character - has a similar purpose: it is used to clearly delimit parts of the code from some other parts of the code.

Automatic Semi-Colon Insertion (ASI)

Interestingly, the browser has a feature known as "Automatic Semi-colon Insertion" - meaning, it does a pretty good job of "filling in the blanks" in case there is a missing semi-colon where there should be one.

Effectively, what that means for developers is that most of the time, it makes no difference if a semi-colon is added or not, since the browser is likely to figure it out anyway.

That's why some developers say that you shouldn't bother with adding semi-colons at all.

However, other developers argue that it's better to use it wherever it's needed - for the sake of clarity.

The truth is that most of the time, you can think of adding semi-colons in JavaScript as optional - and somewhat of a stylistic preference.

A note on using the console in the developer tools in your browser

As already mentioned earlier on in this course, one of the reasons why JavaScript is so popular is because it's so approachable.

To get started with JavaScript, all you need is a browser. In this course I'll be using Google Chrome. Once you've installed the browser and run it, right-click on the currently active web page and click the **Inspect** command on the right-click contextual menu.

This will open the Developer Tools and then you can click on the Console tab to open the console, or alternatively, pressing the `esc` key will toggle on and off the console regardless of the currently active Developer Tools panel.

You can type any JavaScript command you like into the DevTools console.

If you need to type multiple lines of code before you run them, make sure to press the SHIFT + ENTER shortcut key to get onto the next line.

Notice the distinction between pressing the `ENTER` key to run the JavaScript code you've typed, versus pressing the `SHIFT + ENTER` shortcut to move onto the next line of code (rather than running the code you've already typed up).

This is all that you need to get started writing JavaScript code!

In the upcoming lessons, feel free to follow along in either of two ways:

1. Using the VS Code editor and the Code Runner extension as previously described
2. Using the Chrome browser itself, and running the code inside the DevTools console as described in this reading

Output a greeting into the console

Now that you know how to get to the Developer Tools' Console tab, let's now use it to run your first piece of real JavaScript code.

In Chrome, with the Developer Tools open, click into the empty space in the console tab, just to the right of the blue `>` character. You should see a blinking vertical line (also referred to as "the cursor"). The cursor indicates that you can type into the console.

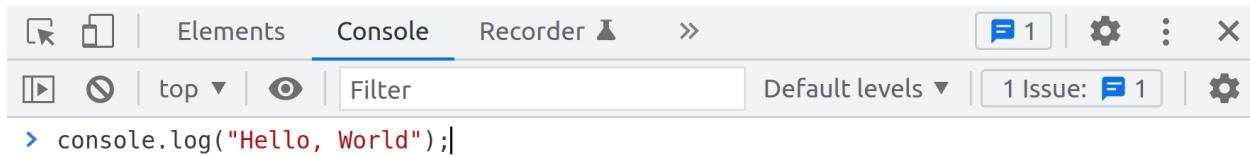
If you type valid JavaScript code, it will be executed, meaning: it will be processed and it might result in some kind of output.

You'll now use the `console.log` function to output the words "Hello, World".

To do so, type the following command into the console:

1

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



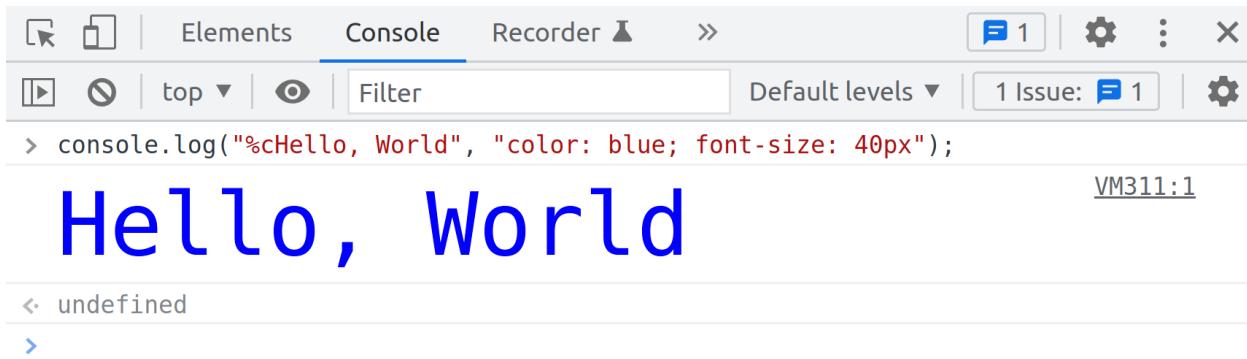
If you've done everything as instructed, the words "Hello, World" should be output in the console. Here's another, more complex command, to show you that the `console.log` command comes with a number of tricks.

For example, did you know that you can style the output in the console?

In this code snippet, there are a few additions: the font size is different and the color is blue:

1

```
console.log("%cHello, World", "color: blue; font-size: 40px");
```



```
> console.log("%cHello, World", "color: blue; font-size: 40px");
Hello, World
VM311:1
```

undefined

console log with percentage c authors

If you copy-paste this piece of code, or perhaps, simply type it into the console, once you press the **ENTER** key to run it, you'll get the words "Hello, World" output to the console. This time, however, the color of the letters will be blue, and the font size will be 40px. So, you've just learned a nice trick with the console.

If you add the `%c` right after the `"` character, you can then style the console output by adding the `,` character after the second `",`, and then, inside another pair of `"` and `"` characters, use valid CSS code to style the words you want to output in the console.

The reason for showing you this little trick was to hopefully get you motivated to practice writing various words into the `console.log` command, and to use your CSS skills to change the styling of these words in the console output. That way, you might find it fun to practice this newly acquired skill - and learning and fun always go nicely together.

Output multiple words into the console

To output multiple words into the console, you can join them using the `+` character, formally known as the "concatenation operator" when we're joining pieces of text, or the "addition operator", for performing the mathematical operation of adding two numbers.

Here is an example of joining three separate pieces of text: `console.log("Hello " + "there, " + "World")`. The output of this command will be: *Hello there, World*.

Here is an example of outputting three separate pieces of text using the `,` character instead: `console.log("Hello ", "there, ", "World")`

The output of this command will still be: *Hello there, World*.

Completed

Exercise: Declaring variables

In this exercise, you will practice declaring variables.

To check the output of your code, please enter it into the text box provided and click the "Run" button. This will execute the code and display the resulting output.

Tasks

1. Declare a new variable named `petDog` and give it the name `Rex`.
2. Declare a new variable named `petCat` and give it the name `Pepper`.
3. `Console.log` the `petDog` variable.
4. `Console.log` the `petCat` variable.
5. `Console.log` the text "`My pet dog's name is:` " and the `petDog` variable.
6. `Console.log` the text "`My pet cat's name is:` " and the `petCat` variable.
7. Declare another variable and name it `catSound`. Assign the string of "`purr`" to it.
8. Declare another variable and name it `dogSound`. Assign the string of "`woof`" to it.
9. `Console.log` the variable `petDog`, then the string "`says`", then the variable `dogSound`.
10. `Console.log` the variable `petCat`, then the string "`says`", then the variable `catSound`.
11. Reassign the value stored in `catSound` to the string "`meow`".
12. `Console.log` the variable `petCat`, then the string "`now says`", then the variable `catSound`.

Make sure to output all your variables. Feel free to play.

1

[Run](#)

[Reset](#)

Completed

Declaring variables (solutions)

Here are the solutions to the previous exercise, which was to practice declaring variables.

Please note: The solutions presented here use single quotes and double quotes interchangeably to delimit pieces of text. This is done on purpose to illustrate that both ways of representing text are possible and acceptable so that you are aware of it.

Task 1 solution to: Declare a new variable named `petDog` and give it the name `Rex`.

1

```
var petDog = 'Rex';
```

Run

Reset

Task 2 solution to: Declare a new variable named `petCat` and give it the name `Pepper`.

1

```
var petCat = 'Pepper';
```

Run

Reset

Task 3 solution to: Console log the `petDog` variable.

1

2

3

```
var petDog = 'Rex'; // Task 1 solution  
console.log(petDog);
```

Run

Reset

Task 4 solution to: Console log the `petCat` variable.

1

2

3

```
var petCat = 'Pepper'; // Task 2 solution  
  
console.log(petCat);
```

Run

Reset

Task 5 solution to: `Console.log` the string "My pet dog's name is: " and the `petDog` variable.

1

2

```
var petDog = 'Rex'; // Task 1 solution  
  
console.log("My pet dog's name is: " + petDog);
```

Run

Reset

Please note that in this specific example, because the text in line 2 contains a single quote within it, you should use double quotes to surround the whole piece of text. Otherwise, JavaScript will return an error. We will cover this issue in more detail later.

Task 6 solution to: `Console.log` the string "My pet cat's name is: " and the `petCat` variable.

1

2

```
var petCat = 'Pepper'; // Task 2 solution  
  
console.log("My pet cat's name is: " + petCat);
```

[Run](#)

[Reset](#)

Please note that in this specific example, because the text in line 2 contains a single quote within it, you should use double quotes to surround the whole piece of text. Otherwise, JavaScript will return an error. We will cover this issue in more detail later.

Task 7 solution to: Declare another variable and name it catSound. Assign the string of "purr" to it.

1

```
var catSound = "purr";
```

[Run](#)

[Reset](#)

Task 8 solution to: Declare another variable and name it dogSound. Assign the string of "woof" to it.

1

```
var dogSound = "woof";
```

[Run](#)

[Reset](#)

Task 9 solution to: Console.log the variable petDog, then the string "says", then the variable dogSound.

1

2

3

```
var petDog = 'Rex'; // Task 1 solution  
  
var dogSound = "woof"; // Task 8 solution  
  
console.log(petDog, "says", dogSound);
```

Run

Reset

Task 10 solution to: Console.log the variable petCat, then the string "says", then the variable catSound.

1

2

3

```
var petCat = 'Pepper'; // Task 2 solution  
  
var catSound = "purr"; // Task 7 solution  
  
console.log(petCat, "says", catSound);
```

Run

Reset

Task 11 solution to: Reassign the value stored in catSound to the string "meow".

1

2

```
var catSound = "purr"; // Task 7 solution  
  
catSound = "meow";
```

Run

[Reset](#)

Task 12 solution to: `Console.log` the variable `petCat`, then the string "now says", then the variable `catSound`.

[1](#)

[2](#)

[3](#)

```
var petCat = 'Pepper'; // Task 2 solution  
  
var catSound = "meow"; // Task 11 solution  
  
console.log(petCat, "now says", catSound);
```

[Run](#)

[Reset](#)

Completed

Operators in depth

In this reading, you will learn about additional operators, operator precedence and operator associativity. I'll also provide you with some examples of logical operators.

1. Additional operators

- Logical AND operator: `&&`
- Logical OR operator: `||`
- Logical NOT operator: `!`
- The modulus operator: `%`
- The equality operator: `==`
- The strict equality operator: `===`
- The inequality operator: `!=`
- The strict inequality operator: `!==`

- The addition assignment operator: `+=`
 - The concatenation assignment operator: `+=` (it's the same as the previous one - more on that later)

The logical AND operator in JavaScript: &&

The **logical AND operator** is, for example, used to confirm if multiple comparisons will return true. In JavaScript, this operator consists of two ampersand symbols together: `&&`.

Let's say you're tasked with coming up with some code that will check if the `currentTime` variable is between 9 a.m. and 5 p.m. The code needs to `console.log true` if `currentTime > 9` and if `currentTime < 17`.

Here's a solution:

1

2

```
var currentTime = 10;  
  
console.log(currentTime > 9 && currentTime < 17);
```

How does this code work?

First, on line one, I set the `currentTime` variable, and assign the value of 10 to it.

Next, on line two I console log two comparisons:

```
currentTime > 9  
currentTime < 1'
```

I also use the `&&` logical operator to join the two comparisons.

Effectively, my code is interpreted as the following:

1

```
console.log(10 > 9 && 10 < 17);
```

The comparison of `10 > 9` will return `true`.

Also, the comparison of `10 < 17` will return `true`.

This means I can further re-write the line two of my solution as follows:

1

```
console.log(true && true);
```

In essence, this is how my code works.

Now, the question is, what will be the result of `console.log(true && true)`? To understand the answer, you need to know the behavior of the `&&` logical operator. The `&&` logical operator returns a single value: the boolean `true` or `false`, based on the following

- It returns `false` in all the other instances

In other words:

```
console.log(true && true) will output: true
console.log(true && false) will output: false
console.log(false && true) will output: false
console.log(false && false) will output: false
```

The logical OR operator in JavaScript: ||

The logical OR operator in JavaScript consists of two pipe symbols together: `||`.

It is used when you want to check if at least one of the given comparisons evaluates to `true`.

Consider the following task: You need to write a program in JavaScript which will return `true` if the value of the `currentTime` variable is not between 9 and 17. Put differently, your code needs to `console.log true` if the value of the variable `currentTime` is either less than 9 or greater than 17.

Here's a solution:

1

2

```
var currentTime = 7;

console.log(currentTime < 9 || currentTime > 17);
```

In line one of the code I assign the number 7 to the variable `currentTime`.

On line two, I `console.log` the result of checking if either `currentTime < 9` or `currentTime > 17` will evaluate to `true`.

It's the same as this:

1

2

```
var currentTime = 7;

console.log(true || false);
```

Here are the rules of how the `||` operator evaluates given values:

```
console.log(true || true) will output: true
console.log(true || false) will output: true
console.log(false || true) will output: true
console.log(false || false) will output: false
```

The logical OR operator will always return `true`, except when both sides evaluate to `false`. In other words, for the logical OR operator to return `false`, the results of both comparisons **must** return `false`.

Going back to the example of checking if either `currentTime < 9` or `currentTime > 17`, this makes sense: the only time you will get `false` is when the value stored in the `currentTime` variable is greater than `9` and less than `17`.

The logical NOT operator: !

In JavaScript, the logical NOT operator's symbol is the exclamation mark: `!`.

You can think of the `!` operator as a switch, which flips the evaluated boolean value from `true` to `false` and from `false` to `true`.

For example if I assign the boolean value of `true` to the `petHungry` variable:

```
var petHungry = true;
```

...then I can console log the fact that the pet is no longer hungry by using the `!` operator to flip the boolean value stored inside of the `petHungry` variable, like so:

```
console.log('Feeding the pet'); console.log("Pet is hungry: ", !petHungry);  
console.log(petHungry);
```

This is the output of the above code:

1

2

3

4

Pet is hungry: `true`

Feeding the pet

Pet is hungry: `false`

`true`

The reason for the changed output in the console is because you have flipped the value stored inside the `petHungry` variable, from `true` to `false`.

Notice, however, that the code on line five of the example above still outputs `true` - that's due to the fact that I didn't reassign the value of the `petHungry` variable.

Here's how I could permanently change the value stored in the `petHungry` variable from `true` to `false`:

1

2

```
var petHungry = true;
```

```
petHungry = !petHungry;
```

In this example, I first assign the value of `true` to the new variable of `petHungry`. Then, on line two, I assign the opposite value, the `!true` - read: not true - to the existing `petHungry` variable.

The modulus operator: %

The modulus operator is another mathematical operator in JavaScript. It returns the remainder of division.

To demonstrate how it works, imagine that a small restaurant that has 4 chairs per table, and a total of 5 tables, suddenly receives 22 guests.

How many guests will not be able to sit down in the restaurant?

You can use the modulus operator to solve this.

1

```
console.log(22 % 5); // 2
```

The output is 2, meaning, when I divide 22 and 5, I get a 4, and the remainder is 2, meaning, there are 2 people who couldn't get a place in this restaurant.

The equality operator, ==

The equality operator checks if two values are equal.

For example, this comparison returns `true`: `5 == 5`. Indeed, it is true that 5 is equal to 5.

Here's an example of the equality operator returning `false`: `5 == 6`. Indeed, it is true that 5 is not equal to 6.

Additionally, even if one of the compared values is of the number type, and the other is of the string type, the returned value is still `true`: `5 == "5"`.

This means that the equality operator compares only the values, but not the types.

The strict equality operator, ===

The strict equality operator compares for both the values and the data types.

With the strict equality operator, comparing `5 === 5` still returns `true`. The values on each side of the strict equality operator have the same value and the same type. However, comparing `5 == "5"` now returns `false`, because the values are equal, but the data type is different.

The inequality operator, !=

The inequality operator checks if two values are not the same, but it does not check against the difference in types.

For example, `5 != "5"` returns false, because it's false to claim that the number 5 is not equal to number 5, even though this other number is of the string data type.

The strict inequality operator !==

For the strict inequality operator to return `false`, the compared values have to have the same value and the same data type.

For example, `5 !== 5` returns `false` because it is false to say that the number 5 is not of the same value and data type and another number 5.

However, comparing the number 5 and the string 5, using the strict inequality operator, returns `true`.

[1](#)

```
console.log(5 !== "5")
```

2. Using the + operators on strings and numbers

Combining two strings using the + operator

The + operator, when used with number data type, adds those values together.

However, the + operator is also used to join string data type together.

For example:

[1](#)

[2](#)

```
"inter" + "net" // "internet"
```

```
"note" + "book" // "notebook"
```

If the + operator is used to join strings, then it is referred to as the *concatenation* operator, and you'll say that it's used to *concatenate* strings.

When used with numbers, the + operator is the **addition operator**, and when used with strings, the + operator is the **concatenation operator**.

Combining strings and numbers using the + operator

But what happens when one combines a string and a number using the + operator?

Here's an example:

[1](#)

[2](#)

[3](#)

```
365 + " days" // "365 days"
```

```
12 + " months" // "12 months"
```

Here, JavaScript tries to help by converting the numbers to strings, and then **concatenating the number and the string together**, ending up with **a string value**.

The process of this "under-the-hood" conversion of values in JavaScript is referred to as "coercion". JavaScript *coerces* a number value to a string value - so that it can run the + operator on disparate data types.

The process of coercion can sometimes be a bit unexpected.

Consider the following example:

1

```
1 + "2"
```

What will be the result of 1 + "2"?

Note that the value of 1 is of the number data type, and the value of "2" is of the string data type, and so JavaScript will coerce the number 1 to a string of "1", and then concatenate it with the string of "2", so the result is a string of "12".

The addition assignment operator, +=

The addition assignment operator is used when one wants to accumulate the values stored in a variable.

Here's an example: You are counting the number of overtime hours worked in a week.

You don't have to specify the type of work, you just want to count total hours.

You might code a program to track it, like this:

1

2

3

4

5

6

```
var mon = 1;  
  
var tue = 2;  
  
var wed = 1;  
  
var thu = 2;
```

```
var fri = 3;  
  
console.log(mon + tue + wed + thu + fri); // 9
```

You can simplify the above code by using the addition assignment operator, as follows:

```
1  
2  
3  
4  
5  
6  
  
var overtime = 1;  
  
overtime += 2;  
  
overtime += 1;  
  
overtime += 2;  
  
overtime += 3;  
  
console.log(overtime); // 9
```

Using the addition assignment operator reduces the lines of your code.

The concatenation assignment operator, +=

This operator's syntax is exactly the same as the addition assignment operator. The difference is in the data type used:

```
1  
2  
3  
4
```

5

6

7

```
var longString = "";  
  
longString += "Once";  
  
longString += " upon";  
  
longString += " a";  
  
longString += " time";  
  
longString += "...";  
  
console.log(longString); // "Once upon a time..."
```

Operator precedence and associativity

Operator precedence is a set of rules that determines which operator should be evaluated first. Consider the following example:

1

```
1 * 2 + 3
```

The result of the above code is 5, because the multiplication operator has precedence over the addition operator.

Operator associativity determines how the precedence works when the code uses operators with the same precedence.

There are two kinds:

- left-to-right associativity
- right-to-left associativity

For example, the assignment operator is right-to-left associative, while the greater than operator is left-to-right associative:

1

2

```
var num = 10; // the value on the right is assigned to the variable name  
on the left
```

```
5 > 4 > 3; // the 5 > 4 is evaluated first (to `true`), then true > 3 is  
evaluated to `false`, because the `true` value is coerced to `1`
```

Completed

Exercise: Advanced use of operators

Task 1: Using the logical && operator

You are coding an RPG game, where each character has certain skill levels based on the value saved in their score.

1. Create a variable named `score` and set it to `8`
2. Use `console.log()` that includes the string "`Mid-level skills:`" and compares the `score` variable to above `0` and below `10` using the `&&` operator

The expected output in the console should be: "`Mid-level skills: true`".

1

Run

Reset

Task 2: Using the logical || operator

Imagine you are coding a video game. Currently, you're about to code some snippets related to the game over condition.

You need to code a new variable named `timeRemaining` and set it to `0`. You also need to code a new variable named `energy` and set it to `10`.

Next, you should write a piece of code that could be used to determine if the game is over, based on whether either the value of the `timeRemaining` variable is `0` or the value of the `energy` variable is `0`.

Complete the task using the following steps:

1. Declare the variable `timeRemaining`, and assign the value of `0` to it.
2. Declare the variable `energy`, and assign the value of `10` to it.
3. Console log the following parameters: "`Game over:` ", and `timeRemaining == 0 || energy == 0`

Note that the expected output in the console should be: "`Game over: true`".

[Run](#)[Reset](#)

Try changing the `timeRemaining` variable to anything above 0 and then see how it affects the result.

Task 3: Using the modulus operator, %, to test if a given number is odd

You need to code a small program that takes a number and determines if it's an even number (like 2, 4, 6, 8, 10).

To achieve this task, you need to declare six variables, as follows:

1. The first variable, named `num1`, should be assigned a number value of 2.
2. The second variable, named `num2`, should be assigned a number value of 5.
3. The third variable, named `test1`, should be assigned the calculation of `num1 % 2`. **Note:** executing this code will return a number.
4. The fourth variable, named `test2`, should be assigned the calculation of `num2 % 2`. **Note:** executing this code will also return a number.
5. The fifth variable, named `result1`, should be assigned the result of comparing if the number stored in the `test1` variable is not equal to 0, in other words, this: `test1 == 0`.
6. The sixth variable, named `result2`, should be assigned the result of comparing if the number stored in the `test2` variable is not equal to 0, in other words, `test2 == 0`.

Run console log two times after you've set the variables:

1. The first console log should have the following code between parentheses: "Is", `num1`, "an even number?", `result1`
2. The second console log should have the following code between parentheses: "Is", `num2`, "an even number?", `result2`

Note: The output to the console should be as follows:

```
Is 2 an even number? true
Is 5 an even number? false
```

[Run](#)[Reset](#)

Try it yourself with different values to explore the modulus operator.

Task 4: Add numbers using the + operator

Console log the result of adding two numbers, 5 and 10, using the + operator.

Note: This task should be completed on a single line of code. The output in the console should be 15.

1

Run

Reset

Task 5: Concatenate numbers and strings using the + operator

Code three variables:

1. The first variable should be a string with the following value: "Now in ". Name the variable `now`.
2. The second variable should be a number with the value: 3. Name the variable `three`.
3. The third variable should a string with the following value: "D!". Name the variable `d`.
4. Console log the following code: `now + three + d`.

Note: The expected output should be: "Now in 3D!".

1

Run

Reset

Task 6: Use the += operator to accumulate values in a variable

Code a new variable and name it `counter`, assigning it to the value of 0.

On the next line, use the += operator to increase the value of counter by 5.

On the next line, use the += operator to increase the value of counter by 3.

On the fourth line, console log the value of the `counter` variable.

Note: The output value should be 8.

1

2

[Run](#)

[Reset](#)

Completed

Advanced use of operators (solutions)

Task 1 Solution

```
1  
2  
3  
  
var score = 8;  
  
console.log("Mid-level skills:", score > 0 && score < 10)
```

[Run](#)

[Reset](#)

Task 2 Solution

```
1  
2  
3  
4  
  
var timeRemaining = 0;  
  
var energy = 10;
```

```
console.log("Game over:", timeRemaining == 0 || energy == 0);
```

Run

Reset

Task 3 Solution

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
  
var num1 = 2;  
  
var num2 = 5;  
  
var test1 = num1 % 2;  
  
var test2 = num2 % 2;  
  
var result1 = test1 == 0;  
  
var result2 = test2 == 0;  
  
console.log("Is", num1, "an even number?", result1);  
console.log("Is", num2, "an even number?", result2);
```

[Run](#)

[Reset](#)

Task 4 Solution

[1](#)

[2](#)

```
console.log(5 + 10); // 15
```

[Run](#)

[Reset](#)

Task 5 Solution

[1](#)

[2](#)

[3](#)

[4](#)

```
var now = "Now in ";
var three = 3;
var d = "D!"

console.log(now + three + d); // "Now in 3D!"
```

[Run](#)

[Reset](#)

Task 6 Solution

[1](#)

2

3

4

5

```
var counter = 0;  
  
counter += 5;  
  
counter += 3;  
  
console.log(counter); // 8
```

[Run](#)

[Reset](#)

Completed

JavaScript improvements

In this reading, you will learn about the history of JavaScript and the importance of ECMA (European Computer Manufacturers Association) and ECMAScript.

JavaScript is a programming language that had humble beginnings.

It was built in only 10 days in 1995 by a single person, Brendan Eich, who was tasked with building a simple scripting language to be used in version 2 of the Netscape browser. It was initially called LiveScript, but since the Java language was so popular at the time, the name was changed to JavaScript - although Java and JavaScript are in no way related.

For the first few years, after it was built, JavaScript was a simple scripting language to add mouseover effects and other interactivity. Those effects were being added to webpages using the `<script>` HTML element.

Inside each of the script elements, there could be some JavaScript code. Due to the rule that HTML, CSS, and JavaScript must be backward compatible, even the most advanced code written in JavaScript today ends up being written between those script tags.

Over the years, JavaScript grew ever more powerful, and in recent times, it's continually touted as among the top three commonly used languages.

In 1996 Netscape made a deal with the organization known as ECMA (European Computer Manufacturers Association) to draft the specification of the JavaScript language, and in 1997 the first edition of the ECMAScript specification was published.

ECMA publishes this specification as the ECMA-262 standard.

You can think of a standard as an agreed-upon way of how things should work. Thus, ECMA-262 is a standard that specifies how the JavaScript language should work.

There have been 12 ECMA-262 updates - the first one was in 1997.

JavaScript as a language is not a completely separate, stand-alone entity. It only exists as an implementation. This implementation is known as a JavaScript engine.

Traditionally, the only environment in which it was possible to run a JavaScript engine, was the browser. More specifically, a JavaScript engine was just another building block of the browser. It was there to help a browser accomplish its users' goal of utilizing the internet for work, research, and play.

So, when developers write JavaScript code, they are using it to interact with a JavaScript engine.

Put differently, developers write JavaScript code so that they can "talk to" a JavaScript engine.

Additionally, the JavaScript engine itself comes with different ways to interact with various other parts of the browser. These are known as Browser APIs.

Thus, the code that you write in the JavaScript programming language allows you to: 1. Interact with the JavaScript engine inside of the browser 2. Interact with other browser functionality that exists outside of the JavaScript engine, but is still inside the browser.

Although traditionally it was possible to interact with the JavaScript engine only inside of the browser, this all changed in 2009, when Node.js was built by Ryan Dahl.

He came up with a way to use a JavaScript engine as a stand-alone entity. Suddenly, it was possible to use JavaScript outside of the browser, as a separate program on the command line, or as a server-side environment.

Today, JavaScript is ubiquitous and is running in browsers, on servers, actually, on any device that can run a JavaScript engine.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

These resources provide some more in-depth information on the topics covered in this module.

[Mozilla Developer Network Expressions and Operators](#)

[Mozilla Developer Network Operator Precedence and Associativity](#)

[JavaScript Primitive Values](#)

[ECMA262 Specification](#)

[jQuery Official Website](#)

[React Official Website](#)

[StackOverflow Developer Survey 2021 Most Popular Technologies](#)

[Emojis](#)

Completed

Conditional examples

In this reading, you will learn when to use the `if else` statement and when to use the `switch` statement.

Both `if else` and `switch` are used to determine the program execution flow based on whether or not some conditions have been met.

This is why they are sometimes referred to as **flow control statements**. In other words, they control the flow of execution of your code, so that some code can be skipped, while other code can be executed.

At the heart of both flow control structures lies the evaluation of one or more conditions.

Generally, `if else` is better suited if there is a binary choice in the condition.

For example, in plain English: *if it's sunny, wear sunglasses. Otherwise, don't.*

In this case, using an `if` statement is an obvious choice.

When there are a smaller number of possible outcomes of truthy checks, it is still possible to use an `if else` statement, such as:

1

2

3

4

5

6

7

8

9

10

```
if(light == "green") {
```

```
console.log("Drive")

} else if (light == "orange") {

    console.log("Get ready")

} else if (light == "red") {

    console.log("Dont' drive")

} else {

    //this block will run if no condition matches

    console.log("The car is not green, orange, or red");

}
```

However, if there are a lot of possible outcomes, it is best practice to use a switch statement because it is easier less verbose. Being easier to read, it is easier to follow the logic, and thus reduce cognitive load of reading multiple conditions.

Nevertheless, this is not a rule set in stone. It is simply a stylistic choice.

To reinforce this point, here's an example of the earlier `if` `else` conditional statement, using the switch syntax:

1
2
3
4
5
6
7
8
9

```
10  
11  
12  
13  
14  
15  
16  
  
//converting the previous if-else example with switch-case  
  
switch(light) {  
  
    case 'green':  
  
        console.log("Drive");  
  
        break;  
  
    case 'orange':  
  
        console.log("Get ready");  
  
        break;  
  
    case 'red':  
  
        console.log("Don't drive");  
  
        break;  
  
    default:  
  
        //this block will run if no condition matches  
  
        console.log('The light is not green, orange, or red');
```

```
    break;  
}  
  
Completed
```

Exercise: Practice conditional statements

Introduction

In this exercise, you will practice working with `if` `else` statements. By the end of this exercise, you will be able to write an `if` `else` statement that determines your source of income based on your age. You will also be able to write a `switch` statement that determines your evening routine based on the day of the week.

Complete the following steps to create: Are You Old Enough?

1. Declare a variable `age` using the `var` keyword and set it to the number **10**.
2. Add an `if` statement that checks if the value of the `age` variable is greater than or equal to the number **65**. Inside the `if` block, `console.log` the sentence: "You get your income from your pension".
3. Add an "`else if`", where you'll check if the value of the `age` is less than **65** and greater than or equal to **18**. Inside this "`else if`" block, type "`console.log`" and then "Each month you get a salary".
4. Add another "`else if`", and this time check if the value of the `age` is under **18**. Inside the "`else if`" block, "type `console.log`" and then "You get an allowance".
5. Add an "`else`" statement to capture any other value. Inside the block, type "`console.log`" and then "The value of the `age` variable is not numerical".

Try adjusting the `age` and executing the program to see how it will affect the output.

Code the *days of the week* program as a switch statement

1. On the next line, define a new variable, name it `day`, and set its value to "`Sunday`".
2. Start coding a `switch` statement, passing the `day` variable as the expression to evaluate.
3. Inside the `switch`, add cases for every day of the week, starting with '`Monday`', and ending with '`Sunday`'. Make sure to use string values for days. Inside each case, for now, just add a `console.log('Do something')`, and add a `break`; on the line below.
4. At the very bottom of the `switch` statement, add the default case and add a `console.log('There is no such day')`.
5. Finally, update the `console.log` calls for each case, based on whatever activity you have on each of the days.

Tips

- If you need to make sure that multiple conditions are true in an if statement, you can do so using the `&&` operator
- In JavaScript, the correct syntax of the "greater than or equal to" operator is: `>=`.
- Don't forget to add a break at the very end of each case in a switch statement.

Note: You can find solutions in a separate reading (following this one)

[1](#)

[Run](#)

[Reset](#)

Completed

Solutions: Practice conditional statements

Solutions to the task - Are you old enough?

Step 1:

Declare a variable age using the `var` keyword and set it to the number 10.

[1](#)

[2](#)

```
var age = 10;
```

[Run](#)

[Reset](#)

Step 2:

Add an "if" statement that checks if the value of the age variable is greater than or equal to the number 65. Inside the if block, type "console.log" and "You get your income from your pension".

[1](#)

2

3

4

5

```
var age = 10;

if (age >= 65) {

    console.log('You get your income from your pension')

}
```

Run

Reset

Step 3:

Add an "else", followed with an "if", where you'll check if the value of age is less than 65 and greater than or equal to 18. Inside this if block type "console.log" and "Each month you get a salary".

1

2

3

4

5

6

7

```
var age = 10;

if (age >= 65) {
```

```
console.log('You get your income from your pension')

} else if (age < 65 && age >= 18) {

    console.log('Each month you get a salary')

}
```

[Run](#)

[Reset](#)

Step 4:

Add another "else if", and this time check if the value of the age is under 18. Inside the if block, type "console.log" and "You get an allowance".

```
1

2

3

4

5

6

7

8

9

var age = 10;

if (age >= 65) {

    console.log('You get your income from your pension')

} else if (age < 65 && age >= 18) {
```

```
console.log('Each month you get a salary')

} else if (age < 18) {

    console.log('You get an allowance')

}
```

[Run](#)

[Reset](#)

Step 5:

Add an "else" statement to capture any other value. Inside the block, type "console.log" and "The value of the age variable is not numerical".

```
1
2
3
4
5
6
7
8
9
10
11
12

var age = 10;
```

```
if (age >= 65) {  
  
    console.log('You get your income from your pension')  
  
} else if (age < 65 && age >= 18) {  
  
    console.log('Each month you get a salary')  
  
} else if (age < 18) {  
  
    console.log('You get an allowance')  
  
} else {  
  
    //this block will run if no condition matches  
  
    console.log('The value of the age variable is not numerical')  
  
}
```

Run

Reset

Solutions to the *Days of the Week* program

Step 1:

On the next line, define a new variable, name it day, and set its value to Sunday.

1

2

```
var day = `Sunday`;
```

Run

Reset

Step 2:

Start coding a switch statement, passing the day variable as the expression to evaluate.

```
1  
2  
3  
4  
5  
  
var day = `Sunday`;  
  
switch(day) {  
  
    //add your conditions  
  
}
```

Run

Reset

Step 3:

Inside the switch, add cases for all the days of the week, starting with 'Monday', and ending with 'Sunday'. Make sure to use string values for days. Inside each case, for now, just add a console.log('Do something'), and add a break; on the line below.

```
1  
2  
3  
4  
5  
6  
7
```

8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

```
var day = `Sunday`;  
  
switch(day) {  
  case 'Monday':
```

```
    console.log('Do something');

    break;

case 'Tuesday':

    console.log('Do something');

    break;

case 'Wednesday':

    console.log('Do something');

    break;

case 'Thursday':

    console.log('Do something');

    break;

case 'Friday':

    console.log('Do something');

    break;

case 'Saturday':

    console.log('Do something');

    break;

case 'Sunday':

    console.log('Do something');

    break;

}
```

[Run](#)

[Reset](#)

Step 4:

At the very bottom of the switch statement, add the default case and add a "console.log"('There is no such day').

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

```
18
19
20
21
22
23
24
25
26
27
28

var day = `Sunday`;

switch(day) {
    case 'Monday':
        console.log('Do something');

        break;

    case 'Tuesday':
        console.log('Do something');

        break;

    case 'Wednesday':
        console.log('Do something');
```

```
break;

case 'Thursday':
    console.log('Do something');

    break;

case 'Friday':
    console.log('Do something');

    break;

case 'Saturday':
    console.log('Do something');

    break;

case 'Sunday':
    console.log('Do something');

    break;

default:
    //this block will run if no condition matches
    console.log('There is no such day');

}
```

Run

Reset

Step 5:

Finally, update the console.log calls for each case, based on whatever activity you have on each of the days.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

```
var day = `Sunday`;

switch(day) {

    case 'Monday':
        console.log('Read a book');

        break;

    case 'Tuesday':
        console.log('Watch a movie');

        break;

    case 'Wednesday':
        console.log('Read a book');

        break;

    case 'Thursday':
        console.log('Play basketball');
```

```
break;

case 'Friday':
    console.log('Socialize');

    break;

case 'Saturday':
    console.log('Chill');

    break;

case 'Sunday':
    console.log('Have barbecue');

    break;

default:
    //this block will run if no condition matches
    console.log('There is no such day');

}
```

Run

Reset

Completed

Exercise: Repetitive tasks with loops

In this exercise, you will practice writing "for" and "while" loops.

Task 1

Write a "`for`" loop that will perform exactly the same repetitive code as this:

```
1  
2  
3  
4  
5  
6  
7  
  
console.log(1)  
console.log(2)  
console.log(3)  
console.log(4)  
console.log(5)  
console.log('Counting completed!')
```

[Run](#)

[Reset](#)

Task 2

Write a "`for`" loop that will perform exactly the same repetitive code as this:

1

2

3

4

5

6

7

```
console.log(5)  
console.log(4)  
console.log(3)  
console.log(2)  
console.log(1)  
console.log('Countdown finished!')
```

Run

Reset

Task 3

Write a "while" loop that will perform exactly the same repetitive code as this:

1

2

3

4

5

6

```
console.log(1)  
console.log(2)  
console.log(3)  
console.log(4)  
console.log(5)  
console.log('Counting completed!')
```

Run

Reset

Note: Name your increment variable `i`. Update the variable in the while loop using `i++`.

Task 4

Write a "`while`" loop that will perform exactly the same repetitive code as this:

```
1  
2  
3  
4  
5  
6  
7  
  
console.log(5)  
console.log(4)
```

```
console.log(3)  
console.log(2)  
console.log(1)  
console.log('Countdown finished!')
```

[Run](#)

[Reset](#)

Note: In the while loop, decrement the value of `i` using: `i = i - 1`.

Task 5

Write a "`while`" loop that will perform exactly the same repetitive code as this:

1

2

3

4

5

6

```
console.log(2018)  
console.log(2019)  
console.log(2020)  
console.log(2021)  
console.log(2022)
```

[Run](#)

[Reset](#)

Completed

Repetitive tasks with loops (solutions)

Here are the solutions to the `for` and `while` loop exercise.

Task 1

[1](#)

[2](#)

[3](#)

[4](#)

[5](#)

```
for (var i = 1; i <= 5; i++) {  
    console.log(i);  
}  
  
console.log('Counting completed!');
```

[Run](#)

[Reset](#)

Task 2

[1](#)

[2](#)

```
3  
4  
5  
  
for (var i = 5; i > 0; i--) {  
  
    console.log(i);  
  
}  
  
console.log('Countdown finished!');
```

Run

Reset

Task 3

```
1  
2  
3  
4  
5  
6  
7  
  
var i = 1;  
  
while (i < 6) {  
  
    console.log(i);
```

```
i++;  
};  
console.log('Counting completed!');
```

[Run](#)

[Reset](#)

Note: Name your increment variable `i`. Update the variable in the while loop using `i++`.

Task 4

Write a "while" loop that will perform the exact same repetitive code like the one below:

```
var i = 5;  
while (i > 0) {  
    console.log(i);  
    i = i - 1;  
}  
console.log('Counting completed!');
```

1
2
3
4
5
6
7

[Run](#)

[Reset](#)

Note: In the while loop, decrement the value of i using: `i = i - 1`.

Task 5

Write a "while" loop that will perform the exact same repetitive code like the one below:

1

2

3

4

5

6

```
var year = 2018;

while (year < 2023) {

    console.log(year);

    year++;

};
```

[Run](#)

[Reset](#)

Completed

Loops and nested loops

Let's say I want to output a custom multiplication table.

This is a perfect use case scenario for nested loops.

The outer loop's counter variable will act as the first number to be multiplied, and the inner loop counter variable will act as the second number to be multiplied.

Here's my code:

```
1  
2  
3  
4  
  
//single loop  
  
for (var firstNum = 0; firstNum < 2; firstNum++) {  
  
    console.log(firstNum);  
  
}
```

The output of the above code will be:

0 1

This means that my for loop starts at 0 and stops after 1.

So now I can code what will later become the inner loop, whose counter variable will be the second number in this multiplication:

```
1  
2  
3  
4  
  
//single loop
```

```
for (var secondNum = 0; secondNum < 10; secondNum++) {  
  
    console.log(secondNum);  
  
}
```

This time, the output is:

0 1 2 3 4 5 6 7 8 9

Now's the time to combine the first and the second loop:

```
1  
  
2  
  
3  
  
4  
  
5  
  
6  
  
  
//nested loops - one inside another  
  
for (var firstNum = 0; firstNum < 2; firstNum++) {  
  
    for (var secondNum = 0; secondNum < 10; secondNum++) {  
  
        console.log(firstNum + ", " + secondNum);  
  
    }  
  
}
```

Now that I'm nesting the second for loop inside the first one, and that I'm console logging the values of both counter variables as the loops are progressing, the output looks like this:

0, 0 0, 1 0, 2 0, 3 0, 4 0, 5 0, 6 0, 7 0, 8 0, 9 1, 0 1, 1 1, 2 1, 3 1, 4 1, 5 1, 6 1, 7 1, 8 1, 9

Now that I have a list of all the numbers that will be multiplied, having the actual result of this multiplication is as easy as updating the `console.log()` call:

```
1  
  
2
```

3

4

5

6

```
//nested loops - one inside another

for (var firstNum = 0; firstNum < 2; firstNum++) {

    for (var secondNum = 0; secondNum < 10; secondNum++) {

        console.log(firstNum + " times " + secondNum + " equals " +
firstNum * secondNum);

    }

}
```

The output now is:

0 times 0 equals 0 0 times 1 equals 0 0 times 2 equals 0 0 times 3 equals 0 0 times 4 equals 0 0 times 5 equals 0 0 times 6 equals 0 0 times 7 equals 0 0 times 8 equals 0 0 times 9 equals 0 1 times 0 equals 0 1 times 1 equals 1 1 times 2 equals 2 1 times 3 equals 3 1 times 4 equals 4 1 times 5 equals 5 1 times 6 equals 6 1 times 7 equals 7 1 times 8 equals 8 1 times 9 equals 9

This makes for some very interesting combinations.

For example, I can make a custom division table:

1

2

3

4

5

6

```
//nested loops - one inside another
```

```

for (var i = 100; i > 10; i = i - 10) {

    for (var j = 10; j > 4; j = j - 5) {

        console.log(i + " divided by " + j + " equals " + i / j);

    }

}

}

```

Here's the output of the above nested loop:

100 divided by 10 equals 10
 100 divided by 5 equals 20
 90 divided by 10 equals 9
 90 divided by 5 equals 18
 80 divided by 10 equals 8
 80 divided by 5 equals 16
 70 divided by 10 equals 7
 70 divided by 5 equals 14
 60 divided by 10 equals 6
 60 divided by 5 equals 12
 50 divided by 10 equals 5
 50 divided by 5 equals 10
 40 divided by 10 equals 4
 40 divided by 5 equals 8
 30 divided by 10 equals 3
 30 divided by 5 equals 6
 20 divided by 10 equals 2
 20 divided by 5 equals 4

Feel free to try out some other combinations of nested loop iterations, and see what kind of output you'll get.

Completed

Uses of loops

In this reading, we'll discuss, at a very high level, the reasons to use loops in JavaScript.

Note that we will keep this discussion high-level because there are multiple "pieces of the puzzle" that are still missing from your understanding at this point.

This is why we will not get bogged-down in the detail of syntax and implementation, but instead, simply discuss how and why loops are used in everyday work of JavaScript developers.

Consider the following example: You work as a developer for an online store.

The store is selling letter cubes for toddlers, and the entire "Shop now" section of the site is organized in a layout where each cube on sale is displayed in a simple card component, with an image of the cube, the letter it teaches, a short description, and the price.

Cards are organized in rows, so that each row contains three cards - three different letters.

Each card is a preview of that specific letter cube on sale, and it's also a link to an entire page, dedicated to providing more info about the cubes, their teaching value, and providing the visitor with a way to complete their checkout process.

Now, here's a quick question: where would loops fit into displaying this grid of cards showcasing the letter cubes on sale?

To understand just how this works, let me code a basic prototype of how this might work.

Since you still don't have enough knowledge to display website layouts in browser with the help of JavaScript, for now I'll have to settle for using a simple string and the console.
Still, this should be a fun exercise.

1

2

3

4

5

6

```
var cubes = 'ABCDEFG';

//styling console output using CSS with a %c format specifier

for (var i = 0; i < cubes.length; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}
```

Run

Reset

Note: In order to have the styles applied, try running this code snippet in your browser's console.

That's it, with this simple code, the output in the console shows each letter on a separate line, styled like a letter cube for toddlers.

The code itself should be mostly familiar, except for the `cubes.length` and the `cubes[i]` syntax. Without getting into too many details, here are both code snippets explained as simple as possible. The `cubes.length` returns a number. Since `cubes` is a string of characters the `cubes.length` gives me the length of the string saved in the variable.

So this gives me the number 7, effectively making my for loop look like this:

1

2

3

4

5

6

```
var cubes = 'ABCDEFG';

//styling console output using CSS with a %c format specifier

for (var i = 0; i < 7; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid
blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}
```

```
> for (var i = 0; i < 7; i++) {
    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";
    console.log("%c" + cubes[i], styles)
}
A
B
C
D
E
F
G
```

The second piece of code that's new here is the `cubes[i]` snippet.

This simply targets each individual letter in the loop, based on the current value of the `i` variable.

In other words, `cubes[i]`, when `i` is equal to 0, is: `A`.

Then, `cubes[i]`, when `i` is equal to 1, is: `B`.

This goes on for as many loops my for loop runs - and this is determined by the `cubes.length` value.

It's also very versatile, since, if I, for example, decided to change the length of the `cubes` string, I would not have to update the condition of `i < cubes.length`, because it gets automatically updated when I change the length of the `cubes` string.

There are some other ways to store data in JavaScript apps that you haven't heard about.

But we can use the same approach with those other kinds of data, to achieve results that essentially work on the same principle as the one just described.

Using loops is the essence of the approach taken in developing many different pieces of functionality in software today.

Some additional examples

If I'm coding an email client, I will get some structured data about the emails to be displayed in the inbox, then I'll use a loop to actually display it in a nicely-formatted way.

If I'm coding an e-commerce site selling cars, I will get a source of nicely-structured data on each of the cars, then loop over that data to display it on the screen.

If I'm coding a calendar online, I'll loop over the data contained in each of the days to display a nicely-formatted calendar.

There are many, many other examples of using loops in code.

Using loops with data that is properly formatted for a given task is a crucial component of building software.

In the lessons that follow, we'll learn about different ways of grouping related data and of displaying it on the screen using JavaScript.

When combined with what you've already learned about loops, this gives you the skills to build various kinds of user interfaces where there is repetitive information.

Some more specific examples include:

- looping over blog post titles in some structured data, and displaying each blog post title on a blog home page
- looping over social media posts in some structured data, and displaying each social media post based on some conditions
- looping over some structured data on clothing available for sale in an online clothing store, and displaying relevant data for each item of clothing

Now you understand the importance of knowing how to work with loops in JavaScript. In the upcoming lessons, we'll learn other relevant information which will allow you to be able to do this.

Completed

Exercise: Working with conditionals and loops

Exercise 1

In this exercise, you will create the code for a `for` loop, using the counter variable named `i` starting from `1`.

To make the counter increment by `1` on each loop, you will use `i++`.

The exit condition for the for loop should match the output given below.

Inside the loop, write an if-else statement, which will check the following conditions:

1. First, it will check if the value of `i` is `1`. If it is, your code will console log the string "**Gold medal**".
2. Next, I will check if the value of `i` is `2`. If it is, your code will console log the string "**Silver medal**".
3. Then, your code will check if the value of `i` is `3`. If it is, it will console log the string "**Bronze medal**".
4. For all the remaining values of `i`, your code will console log just the value of `i`.

Note: The expected console log of the entire code should be as follows. **Gold medal Silver medal Bronze medal 4 5 6 7 8 9 10**

[1](#)

[Run](#)

[Reset](#)

Exercise 2. Use the completed code from the previous task, but convert the conditionals to a switch statement.

When you code the solution, the output in the console should remain exactly the same as in the previous question.

Note: You'll need three separate cases for the three medals, and a default case for all other values of the `i` variable.

[1](#)

[Run](#)

[Reset](#)

Completed

Solution: Working with conditionals and loops

Answer 1:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
  
for (var i = 1; i <= 10; i++) {  
    if(i == 1) {  
        console.log("Gold medal")  
    } else if (i == 2) {  
        console.log("Silver medal")  
    } else if (i == 3) {  
        console.log("Bronze medal")  
    } else if (i == 4) {  
        console.log("Silver medal")  
    } else if (i == 5) {  
        console.log("Bronze medal")  
    } else if (i == 6) {  
        console.log("Silver medal")  
    } else if (i == 7) {  
        console.log("Bronze medal")  
    } else if (i == 8) {  
        console.log("Silver medal")  
    } else if (i == 9) {  
        console.log("Bronze medal")  
    } else if (i == 10) {  
        console.log("Gold medal")  
    }  
}
```

```
    } else {  
  
        //this block will run if no condition matches  
  
        console.log(i)  
  
    }  
  
}
```

Run

Reset

Answer 2:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

```
for (var i = 1; i <= 10; i++) {  
  
    switch(i) {  
  
        case 1:  
  
            console.log("Gold medal")  
  
            break  
  
        case 2:  
  
            console.log("Silver medal")  
  
            break  
  
        case 3:  
  
            console.log("Bronze medal")  
  
            break  
  
        default:  
  
            //this block will run if no condition matches  
  
            console.log(i)  
  
    }  
}
```

[Run](#)

[Reset](#)

Completed

Additional resources for Conditionals and Loops

Here is a list of resources that may be helpful as you continue your learning journey.

[Comparison Operators](#)

[Truthy](#)

[Falsy](#)

[Conditional statements](#)

In JavaScript, there is also a shorthand version of writing a conditional statement, known as the conditional (ternary) operator: [Conditional \(ternary\) operator](#)

Completed

Building and calling functions

In this reading, you will learn how to build and call a function. The purpose of this reading is to provide you with an example of function declaration (build) and function invocation (call). In the next lesson you will be writing the code.

By the end of this reading you should be able to:

- Code simple functions that can accept an array and iterate through it

Let's start with giving our function declaration a name:

1

2

3

```
function listArrayItems(arr) {  
    // ... code to be added ...  
}
```

So, I've declared a `listArrayItems` function, and I've set it up to accept a single parameter, `arr` - which stands for an array.

Now, I'll need to code a for loop to loop over an array.

As covered in previous lessons in this course, a for loop needs the following information:

1. the starting loop counter value as a temporary variable `i`
2. the exit condition (the maximum value of the loop counter variable `i`, above which the loop no longer runs)
3. how to update the value of `i` after each loop

Here's the information I'll use in this function declaration: 1. The loop's starting counter will be 0. The reason for setting it to zero is due to the fact that arrays are also counted from zero.

This means that I'll have a one-to-one mapping of the current value of the `i` variable at any given time, corresponding to the same index position of any item in the `arr` array 2.

The for loop's exit condition is when the value of `i` is equal or greater than `arr.length`.

Since the `arr.length` counts the number of items in the array from one, and the array items are indexed from zero, this effectively means that as soon as `i` is equal to `arr.length`, the loop will finish and any other code after it will be run.

This practically means that the exit condition for this for loop will be `i < arr.length` returning `false`.

In other words, as long as `i < arr.length` is true, this for loop will continue to run. 3. To make sure that none of the items in the `arr` array are skipped, I have to increase the value of `i` by 1 after each loop.

Now that I know exactly how my for loop should behave, I can add it to my `listArrayItems()` function:

```
function listArrayItems(arr) {  
    for (var i = 0; i < arr.length; i++) {
```

1

2

3

4

5

```
// ... code pending here ...  
}  
  
}
```

Now all that I have left to decide is how I want to output each item from the received `arr` array. It can be as simple as console logging the array item index of the current value of `i`:

```
1  
2  
3  
4  
5  
  
function listArrayItems(arr) {  
  
    for (var i = 0; i < arr.length; i++) {  
  
        console.log(arr[i]) //display the array item where the index is  
        equal to i  
  
    }  
  
}
```

If I now invoke the `listArrayItems` function, I can, for example, give it the following array of colors:

```
1  
2  
  
var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple',  
'pink'];  
  
listArrayItems(colors); //display all items in the array at once
```

The output will be:

1

2

3

4

5

6

7

red

orange

yellow

green

blue

purple

pink

I can update the output any way I like. For example, here are my `arr` items with a number in front of each item:

1

2

3

4

5

6

7

```
//function that takes an array as input and display all items of this
array

function listArrayItems(arr) {

    for (var i = 0; i < arr.length; i++) {

        console.log(i, arr[i])

    }

}

var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple',
'pink'];

listArrayItems(colors);
```

Now the output of the above code will be as follows:

1
2
3
4
5
6
7

0 'red'
1 'orange'
2 'yellow'

```
3 'green'  
4 'blue'  
5 'purple'  
6 'pink'
```

To start the count from one instead of zero, I can update my function declaration as follows:

```
1  
2  
3  
4  
5  
  
function listArrayItems(arr) {  
  
  for (var i = 0; i < arr.length; i++) {  
  
    console.log(i+1, arr[i])  
  
  }  
  
}
```

Invoking the above, updated, function declaration on my `colors` array, will now result in the following output:

```
1  
2  
3  
4  
5  
6
```

```
1 'red'  
2 'orange'  
3 'yellow'  
4 'green'  
5 'blue'  
6 'purple'  
7 'pink'
```

I can even add one or more conditions, such as:

```
1  
2  
3  
4  
5  
6  
7  
8  
9
```

```
function listArrayItems(arr) {  
  
  for (var i = 0; i < arr.length; i++) {  
  
    if (arr[i] == 'red') {
```

```
    console.log(i*100, "tomato!")

} else {

    console.log(i*100, arr[i])

}

}

}
```

Now I'm adding control flow right inside my function, based on whether a specific array member matches a specific value - in this case the string "`red`".

Invoking my newest updated version of the `listArrayItems` function on the `colors` array will now result in the following output:

```
1

2

3

4

5

6

7

0 'tomato!'

100 'orange'

200 'yellow'

300 'green'

400 'blue'

500 'purple'
```

```
600 'pink'
```

Completed

Exercise: Practicing with functions

Your task in this exercise is to code a function which will be able to take a word and locate the position of a chosen letter in that given word.

Task 1:

Write a function named **letterFinder** that accepts two parameters: **word** and **match**.

Task 2:

Code a 'for' loop inside the function's body. The for loop's counter should start at zero, increment by 1 on each iteration and exit when the counter variable's value is equal to the length of the **word** parameter.

Task 3:

Add an if statement inside the for loop whose condition works as follows:

1. Access each of the letters inside the passed in **word** using the counter variable, with **word[i]**.
2. Check if the current **word[i]** is equal to the value of **match**.

Task 4:

console.log the following inside the body of the if statement: **console.log('Found the', match, 'at', i)**.

Task 5:

Write the else condition. Here you'll just console log the following: **console.log('---No match found at', i)**.

Task 6:

Call the **letterFinder** and pass it as its first argument as the string "**test**" and as its second argument, the string "**t**".

Your output should be the following:

Found the t at 0

---No match found at 1

---No match found at 2

Found the t at 3

[Run](#)

[Reset](#)

Completed

Solution: Practicing with functions

Task 1 solution:

1

2

3

4

5

```
// A function that accepts two parameters

function letterFinder(word, match) {

}
```

[Run](#)

[Reset](#)

Task 2 solution:

1

```
2  
3  
4  
5  
6  
  
function letterFinder(word, match) {  
  
    for(i = 0; i < word.length; i++) {  
  
        //this loop exists when i is equal to the length of the word  
  
    }  
  
}
```

Run

Reset

Task 3 solution:

```
1  
2  
3  
4  
5  
6  
7
```

```
function letterFinder(word, match) {  
  
    for(i = 0; i < word.length; i++) {  
  
        if(word[i] == match) {  
  
            //check if the current character, word[i], is equal to the  
            match  
  
        }  
  
    }  
}
```

Run

Reset

Task 4 solution:

1

2

3

4

5

6

7

8

9

```
function letterFinder(word, match) {  
  
    for(i = 0; i < word.length; i++) {  
  
        if(word[i] == match) {  
  
            //check if the current character, word[i], is equal to the  
            match  
  
            console.log('Found the', match, 'at', i)  
  
        }  
  
    }  
  
}
```

Run

Reset

Task 5 solution:

1

2

3

4

5

6

7

8

9

10

11

```
function letterFinder(word, match) {  
  
    for(i = 0; i < word.length; i++) {  
  
        if(word[i] == match) {  
  
            //check if the current character, word[i], is equal to the  
            match  
  
            console.log('Found the', match, 'at', i)  
  
        } else {  
  
            console.log('---No match found at', i)  
  
        }  
  
    }  
  
}  
  
}
```

Run

Reset

Task 6 solution:

1

2

3

4

5

```
6
7
8
9
10
11
12
13

function letterFinder(word, match) {

    for(var i = 0; i < word.length; i++) {

        if(word[i] == match) {

            //check if the current character, word[i], is equal to the
            match

            console.log('Found the', match, 'at', i)

        } else {

            console.log('---No match found at', i)

        }
    }

}

letterFinder("test", "t")
```

[Run](#)

[Reset](#)

Completed

Object Literals and the Dot Notation

By the end of this reading, you'll be able to:

- Explain one of the three common ways to build objects (using the object literal notation)
- Outline the common way to add new properties to objects (or update existing properties) using the dot notation

Object literals and the dot notation

One of the most common ways of building an object in JavaScript is using the object literal syntax:
`{ }.`

To be able to access this object literal, it is very common to assign it to a variable, such as:

[1](#)

```
var user = {};  
//create an object
```

Now an object literal is assigned to the variable `user`, which means that the object it is bound to can be extended and manipulated in a myriad of ways.

Sometimes, an entire object can be immediately built, using the object literal syntax, by specifying the object's properties, delimited as key-value pairs, using syntax that was already covered in an earlier lesson item in this lesson.

Here's one such previously built object:

[1](#)

[2](#)

[3](#)

[4](#)

5

6

7

8

9

```
//creating an object with properties and their values

var assistantManager = {

    rangeTilesPerTurn: 3,

    socialSkills: 30,

    streetSmarts: 30,

    health: 40,

    specialAbility: "young and ambitious",

    greeting: "Let's make some money"

}
```

The beauty of this syntax is that it's so easily readable.

It essentially consists of two steps:

1. Declaring a new variable and assigning an object literal to it - in other words, this: **var assistantManager = {}**

2. Assigning the values to each of the object's keys, using the assignment operator, =

Notice that it's very easy to build any kind of an object in JavaScript using this example syntax.

For example, here's a **table** object:

1

2

3

4

```
var table = {
  legs: 3,
  color: "brown",
  priceUSD: 100,
}
```

To access the `table` object, I can simply console log the entire object:

1

```
console.log(table); //display the object in the developer console
```

The returned value is the entire `table` object:

1

```
{legs: 3, color: 'brown', priceUSD: 100}
```

Additionally, I can console log any individual property, like this:

1

```
console.log(table.color); // 'brown'
```

Now that I have this "syntax recipe", I can build any other object in a similar way:

1

2

3

4

5

```
var house = {
  rooms: 3,
```

```
    color: "brown",  
  
    priceUSD: 10000,  
  
}
```

An alternative approach of building objects is to first save an empty object literal to a variable, then use the dot notation to declare new properties on the fly, and use the assignment operator to add values to those properties; for example:

```
1  
  
2  
  
3  
  
4  
  
var house2 = {};  
  
house2.rooms = 4;  
  
house2.color = "pink";  
  
house2.priceUSD = 12345;
```

Additionally, nothing is preventing me from combining the two approaches. For example:

```
1  
  
2  
  
3  
  
console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000}  
  
house.windows = 10;  
  
console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000,  
windows: 10}
```

This flexibility additionally means that I can update already existing properties, not just add new ones:

1

```
house.windows = 11;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000,
windows: 11}
```

Completed

Object Literals and the Brackets Notation

By the end of this reading, you'll be able to:

- Explain how to build objects using the brackets notation
- Explain that with the brackets notation you can use the space character inside keys, since property keys are strings
- Explain that the keys inside the brackets notation are evaluated

Object literals and the brackets notation

There is an alternative syntax to the dot notation I used up until this point.

This alternative syntax is known as *the brackets notation*.

To understand how it works, it's best to use an example, so I'll go through the process of coding the `house2` object again, in the same way that I did with the dot notation, only this time, I'll use the brackets notation.

1

2

3

4

5

```
var house2 = {};

house2["rooms"] = 4;

house2['color'] = "pink";
```

```
house2["priceUSD"] = 12345;

console.log(house2); // {rooms: 4, color: 'pink', priceUSD: 12345}
```

Note that using the brackets notation, I essentially just wrap each property's key **as a string**, inside either the single or double quotes - just like with regular strings.

Then I wrap the entire property key into an opening and a closing square bracket.

That's essentially all there is to it.

I can both access and update properties on objects using either the dot notation, or the brackets notation, or a combination of both, like in the following example:

```
1

2

3

4

5

6

var car = {};

car.color = "red";

car["color"] = "green";

car["speed"] = 200;

car.speed = 100;

console.log(car); // {color: "green", speed: 100}
```

For the time being, this is probably enough information on object creation.

Before I discuss the topic of arrays and objects, let me just give you another important piece of information about the brackets notation.

With the brackets notation, I can add space characters inside property names, like this:

```
1

2
```

```
car["number of doors"] = 4;  
  
console.log(car); // {color: 'green', speed: 100, number of doors: 5}
```

Additionally, I can add numbers (as the string data type) as property keys:

```
1  
  
2  
  
car["2022"] = 1901;  
  
console.log(car); // {2022: 1901, color: 'green', speed: 100, number of doors: 5}
```

However, doing this is discouraged, due to obvious reasons of having a property key as a number string not really conveying a lot of useful information.

Finally, there's one really useful thing that bracket notation has but is not available in the dot notation: It can evaluate expressions.

To understand what that means, consider the following example:

```
1  
  
2  
  
3  
  
4  
  
5  
  
6  
  
7  
  
8  
  
9  
  
var arrOfKeys = ['speed', 'altitude', 'color'];  
  
var drone = {  
    speed: 100,  
    altitude: 500,  
    color: 'red'  
};  
  
for (let key of arrOfKeys) {  
    console.log(drone[key]);  
}
```

```
speed  
altitude  
color
```

```
drone
```

```
speed: 100,
```

```
    altitude: 200,  
  
    color: "red"  
  
}  
  
for (var i = 0; i < arrOfKeys.length; i++) {  
  
    console.log(drone[arrOfKeys[i]])  
  
}
```

The above code will result in the following output:

1
2
3

100
200
red

Using the fact that brackets notation can evaluate expressions, I accessed the `arrOfKeys[i]` property on the `drone` object.

This value changed on each loop while the for loop was running.

Specifically, the first time it ran, it was evaluated like this:

- The value of `i` was 0
- The value of `arrOfKeys[i]` was `arrOfKeys[0]`, which was "speed"
- Thus, `drone[arrOfKeys[i]]` was evaluated to `drone["speed"]` which is equal to 100

This allowed me to loop over each of the values stored inside the `drone` object, based on each of its properties' keys.

Completed

Arrays are Objects

By the end of this reading, you'll be able to:

- Explain that arrays are objects, with their own built-in properties and methods
- Outline the common way to extend arrays using the `push()` method
- and explain how to trim the last member of an array using the `pop()` method

Arrays are objects

In JavaScript, arrays are objects. That means that arrays also have some built-in properties and methods.

One of the most commonly used built-in methods on arrays are the `push()` and the `pop()` methods. To add new items to an array, I can use the `push()` method:

```
1  
2  
3  
  
var fruits = [];  
  
fruits.push("apple"); // ['apple']  
  
fruits.push('pear'); // ['apple', 'pear']
```

To remove the last item from an array, I can use the `pop()` method:

```
1  
2  
  
fruits.pop();  
  
console.log(fruits); // ['apple']
```

Tying into some earlier lessons in this course, I can now build a function that takes all its arguments and pushes them into an array, like this:

```
1  
2  
3  
4  
5  
  
function makeArray(...args) {  
  return args;  
}  
  
makeArray('a', 'b', 'c') // ['a', 'b', 'c']
```

6

7

```
function arrayBuilder(one, two, three) {  
  
  var arr = [];  
  
  arr.push(one);  
  
  arr.push(two);  
  
  arr.push(three);  
  
  console.log(arr);  
  
}
```

I can now call the `arrayBuilder()` function, for example, like this:

1

```
arrayBuilder('apple', 'pear', 'plum'); // ['apple', 'pear', 'plum']
```

Even better, I don't have to console log the newly built array.

Instead, I can return it:

1

2

3

4

5

6

7

```
function arrayBuilder(one, two, three) {  
  
  var arr = [];
```

```
    arr.push(one);  
  
    arr.push(two);  
  
    arr.push(three);  
  
    return arr;  
  
}
```

Additionally, I can save this function call to a variable.

I can name it anything, but this time I'll use the name: `simpleArr`.

[1](#)

```
var simpleArr = arrayBuilder('apple', 'pear', 'plum');
```

And now I can console log the values stored in `simpleArr`:

[1](#)

```
console.log(simpleArr); // ['apple', 'pear', 'plum']
```

Completed

Math object cheat sheet

JavaScript has handy built-in objects. One of these popular built-in objects is the Math object.

By the end of this reading, you'll be able to:

- Outline the built-in properties and methods of the Math object

Number constants

Here are some of the built-in number constants that exist on the Math object:

- The PI number: `Math.PI` which is approximately 3.14159
- The Euler's constant: `Math.E` which is approximately 2.718
- The natural logarithm of 2: `Math.LN2` which is approximately 0.693

Rounding methods

These include:

- `Math.ceil()` - rounds up to the closest integer
- `Math.floor()` - rounds down to the closest integer
- `Math.round()` - rounds up to the closest integer if the decimal is .5 or above; otherwise, rounds down to the closest integer
- `Math.trunc()` - trims the decimal, leaving only the integer

Arithmetic and calculus methods

Here is a non-conclusive list of some common arithmetic and calculus methods that exist on the `Math` object:

- `Math.pow(2, 3)` - calculates the number 2 to the power of 3, the result is 8
- `Math.sqrt(16)` - calculates the square root of 16, the result is 4
- `Math.cbrt(8)` - finds the cube root of 8, the result is 2
- `Math.abs(-10)` - returns the absolute value, the result is 10
- Logarithmic methods: `Math.log()`, `Math.log2()`, `Math.log10()`
- Return the minimum and maximum values of all the inputs: `Math.min(9, 8, 7)` returns 7, `Math.max(9, 8, 7)` returns 9.
- Trigonometric methods: `Math.sin()`, `Math.cos()`, `Math.tan()`, etc.

Completed

String cheat sheet

By the end of this reading, you'll be able to:

- Identify examples of String functions and explain how to call them

In this cheat sheet, I'll list some of the most common and most useful properties and methods available on strings.

For all the examples, I'll be using either one or both of the following variables:

1

2

```
var greet = "Hello, ";
var place = "World"
```

Note that whatever string properties and methods I demo in the following examples, I could have ran it on those strings directly, without saving them to a variable such as the ones I named `greet` and `place`.

In some of the examples that follow, for the sake of clarity, instead of using a variable name, I'll use the string itself.

All strings have at their disposal several built-in properties, but there's a single property that is really useful: the `length` property, which is used like this:

[1](#)

```
greet.length; // 7
```

To read each individual character at a specific index in a string, starting from zero, I can use the `charAt()` method:

[1](#)

```
greet.charAt(0); // 'H'
```

The `concat()` method joins two strings:

[1](#)

```
"Wo".concat("rl").concat("d"); // 'World'
```

The `indexOf` returns the location of the first position that matches a character:

[1](#)

[2](#)

[3](#)

```
"ho-ho-ho".indexOf('h'); // 0
```

```
"ho-ho-ho".indexOf('o'); // 1
```

```
"ho-ho-ho".indexOf('-'); // 2
```

The `lastIndexOf` finds the last match, otherwise it works the same as `indexOf`.

The `split` method chops up the string into an array of sub-strings:

[1](#)

```
"ho-ho-ho".split("-"); // ['ho', 'ho', 'ho']
```

There are also some methods to change the casing of strings. For example:

1

2

```
greet.toUpperCase(); // "HELLO, "
greet.toLowerCase(); // "hello, "
```

Here's a list of all the methods covered in this cheat sheet:

- `charAt()`
- `concat()`
- `indexOf()`
- `lastIndexOf()`
- `split()`
- `toUpperCase()`
- `toLowerCase()`

Completed

Exercise: Creating arrays and objects

In this exercise lab you will practice creating arrays and objects.

Tasks to complete

1. Create a new empty array literal and assign it to the variable `clothes`.
2. Add 5 of your favorite items of clothing as strings using the `push()` method.
3. Remove the fifth piece of clothing from the array using the `pop()` method.
4. Add a new piece of clothing using the `push()` method.
5. Use `console.log` to show the third item from the `clothes` array in the console.
6. Create a new empty object literal and assign it to the variable `favCar`.
7. Using the dot notation, assign a `color` property to the `favCar` object and give it a string value with the color of your choice.
8. Using the dot notation, assign a `convertible` property to the `favCar` object and give it a boolean value of your choice.
9. Use the console to log the entire `favCar` object.

1

[Run](#)

[Reset](#)

Tips

- Remember to use the object literal syntax: {}.
- Remember to use the array literal syntax: [].

Resources

- Video (Conceptual): Arrays
- Video (Mix): Introduction to Arrays
- Video (Conceptual): Objects
- Video (Mix): Objects

Completed

Creating arrays and objects (solutions)

Answers

Step 1

Create a new empty array literal and assign it to the variable `clothes`.

[1](#)

```
var clothes = [];
```

[Run](#)

[Reset](#)

Step 2

Add 5 of your favorite items of clothing as strings using the `push()` method.

[1](#)

[2](#)

3

4

5

6

```
var clothes = [];  
  
clothes.push('gray t-shirt'); // 1st item of clothing  
  
clothes.push('blue t-shirt'); // 2nd item of clothing  
  
clothes.push('yellow t-shirt'); // 3rd item of clothing  
  
clothes.push('slippers'); // 4th item of clothing  
  
clothes.push('old jeans'); // 5th item of clothing
```

Run

Reset

Step 3

Remove the fifth piece of clothing from the array using the `pop()` method.

1

2

3

4

5

6

7

```
var clothes = [];
```

```
clothes.push('gray t-shirt'); // 1st item of clothing  
  
clothes.push('blue t-shirt'); // 2nd item of clothing  
  
clothes.push('yellow t-shirt'); // 3rd item of clothing  
  
clothes.push('slippers'); // 4th item of clothing  
  
clothes.push('old jeans'); // 5th item of clothing  
  
clothes.pop();
```

Run

Reset

Step 4

Add a new piece of clothing using the `push()` method.

1

2

3

4

5

6

7

8

9

```
var clothes = [];  
  
clothes.push('gray t-shirt'); // 1st item of clothing  
  
clothes.push('blue t-shirt'); // 2nd item of clothing
```

```
clothes.push('yellow t-shirt'); // 3rd item of clothing  
  
clothes.push('slippers'); // 4th item of clothing  
  
clothes.push('old jeans'); // 5th item of clothing  
  
clothes.pop();  
  
clothes.push('green scarf');
```

Run

Reset

Step 5

Use `console.log` to show the third item from the `clothes` array in the console.

1
2
3
4
5
6
7
8
9
10

```
var clothes = [];  
  
clothes.push('gray t-shirt'); // 1st item of clothing
```

```
clothes.push('blue t-shirt'); // 2nd item of clothing  
  
clothes.push('yellow t-shirt'); // 3rd item of clothing  
  
clothes.push('slippers'); // 4th item of clothing  
  
clothes.push('old jeans'); // 5th item of clothing  
  
clothes.pop();  
  
clothes.push('green scarf');  
  
console.log(clothes[2]);
```

[Run](#)

[Reset](#)

Step 6

Create a new empty object literal and assign it to the variable `favCar`.

1

2

```
var favCar = {};
```

[Run](#)

[Reset](#)

Step 7

Using the dot notation, assign a `color` property to the `favCar` object and give it a string value with the color of your choice.

1

2

3

```
var favCar = {};  
  
favCar.color = "red";
```

Run

Reset

Step 8

Using the dot notation, assign a `convertible` property to the `favCar` object and give it a boolean value of your choice.

1

2

3

4

```
var favCar = {};  
  
favCar.color = "red";  
  
favCar.convertible = true;
```

Run

Reset

Step 9

Use the console to log the entire `favCar` object.

1

2

3

4

```
var favCar = {};  
  
favCar.color = "red";  
  
favCar.convertible = true;  
  
console.log(favCar);
```

[Run](#)[Reset](#)[Completed](#)

Object Methods

You might already be familiar with objects in JavaScript.

In this video, you will learn how to design objects as combinations of data and functionality.

As you might already know, an object consists of key-value pairs, known as properties.

We can add new key-value pairs to existing objects using the dot notation and the assignment operator.

1

2

```
var car = {};  
  
car.color = "red"; //update the value of a property of the car object
```

These are known as properties, and can take many data types, including functions.

1

2

3

4

5

6

7

8

```
var car = {};  
  
car.color = "red";  
  
//add a method to the car object so that it can be called as car.turnkey()  
  
car.turnKey = function() {  
  
    console.log('engine running');  
  
}  

```

If the function is a property of an object, it is then referred to as a method.

This is a function that can be accessed only through the JavaScript object that it is a member of. For example, the log method, which belongs to the console object, can only be accessed through the console object.

```
console.log('Hello world');
```

Let's explore this further now. I will create an object using something known as the constructor function.

1

2

3

4

```
var car = {};  
  
car.mileage = 98765;  
  
car.color = "red";  
  
console.log(car);
```

[Run](#)

[Reset](#)

First, I'll build a new object literally named `car`. I type `var`, space, `car`, space, equal sign, space, followed by a set of curly braces, and finally a semicolon.

Now, I'll extend the `car` object by assigning it a property named `mileage`.

When I inspect the object, I can confirm that it contains a `mileage` property set to `98765`.

I want to add another property to the `car` object. This time, I will add a property named `color` and set it to the value of `"red"`.

I can inspect the object again by typing its name into the browser console. So now, when I type `console.log(car)`, I get an object with two properties: the `mileage` property, which is set to `98765`, and the `color` property, set to `"red"`.

Great, now I've added two properties to my object.

Next, I want to add a method to my `car` object. And this method, when called, will output some text to the console.

So, once again, I add another property to my `car` object. After all, a method is just another property of the `car` object. It's just another key-value pair that the car object holds.

What's unique is that the value I'm assigning to it is a function.

1

2

3

4

5

6

7

8

```
var car = {};  
  
car.mileage = 98765;  
  
car.color = "red";  
  
console.log(car);  
  
car.turnTheKey = function() {  
  
    console.log("The engine is running")  
  
}  
  
console.log(car);
```

[Run](#)

[Reset](#)

So, I begin by typing `car` dot `turnTheKey`, equals, and then I type the code for my function. So `function`, open-close parentheses. Then the two curly braces where I will place my code. Finally, inside the curly braces, I type the `console` dot `log` followed by the message "`The engine is running`".

Now I can inspect my `car` object again by typing its name into the `console log` method. This time, it displays that the `car` object contains three properties; the `color` property, the `mileage` property, and the `turnTheKey` property.

Remember that all the key-value pairs in an object are referred to simply as properties. However, if I want to differentiate between the properties that can be executed, then I refer to such properties as methods.

So, now I want to add another method to the `car` object. I'll name this one `lightsOn`.

Once again, I type `car.lightsOn`, and then I add an equals sign, and again since it's a method, I'm assigning it to a function. This function will also have a `console log` in its body, and I'm just logging the string with the text "`The lights are on`".

1

2

3

4

5

```
6
7
8
9
10
11
12
13
14

//example of adding properties and methods to an object

var car = {};
car.mileage = 98765;
car.color = "red";
console.log(car);
car.turnTheKey = function() {
    console.log("The engine is running")
}
car.lightsOn = function() {
    console.log("The lights are on.")
}
console.log(car);
```

```
car.turnTheKey();  
  
car.lightsOn()
```

[Run](#)

[Reset](#)

Ok, so now I have added four properties to my object. And two of those are methods. I've already ensured that I'm getting the correct `mileage` and `color` from my `car` object. Now, I'll try executing the `turnTheKey` and the `lightsOn` methods.

First, I'll invoke the `turnTheKey` method.

Remember that this method can be accessed only through the `car` object, so I first need to type the name of the object that holds the `turnTheKey` method. In other words, I need to type the word `car`, followed by a dot, and then the name of my method, which is `turnTheKey`.

Remember that this property is a method. So, to run it, I need to append an opening and a closing parenthesis so that the JavaScript engine can process my JavaScript code.

Notice that this results in the "`The engine is running`" string logged to the console.

Now I'll test the other method. Once again, I need to access it through the `car` object, so I type `car.lightsOn`, and again, I need to add those parentheses to invoke the `lightsOn` method. I press the ENTER key and notice the text displays in the console.

Success! It's important to remember that when the JavaScript engine runs this line of code, it locates the `car` object in its memory. Then, it finds the `lightsOn` method on the `car` object. It reads the function declaration that's saved on this property and runs it, line by line.

Since there's only a single line of code, the JavaScript engine logs the string "`The lights are on`" to the console.

[Completed](#)

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[JavaScript Functions](#)

[JavaScript Object Basics](#)

[typeof operator in JavaScript](#)

[Arrays are "list-like objects"](#)

[Completed](#)

Syntax, logical and runtime errors

By the end of this reading, you'll be able to:

- Recognize common types of errors in JavaScript

Here are some of the most common errors in JavaScript:

- ReferenceError
- SyntaxError
- TypeError
- RangeError

There are some other errors in JavaScript. These other errors include:

- AggregateError
- Error
- InternalError
- URIError

However, in this reading I'll focus on the Reference, Syntax, Type, and Range errors.

ReferenceError

A ReferenceError gets thrown when, for example, one tries to use variables that haven't been declared anywhere.

An example can be, say, attempting to console log a variable that doesn't exist:

1

```
console.log(username);
```

If the variable named `username` hasn't been declared, the above line of code will result in the following output:

1

```
Uncaught ReferenceError: username is not defined
```

SyntaxError

Any kind of invalid JavaScript code will cause a SyntaxError.

For example:

1

```
var a "there's no assignment operator here";
```

The above line of code will throw the following error:

1

```
Uncaught SyntaxError: Unexpected string
```

There's an interesting caveat regarding the `SyntaxError` in JavaScript: it cannot be caught using the `try-catch` block.

TypeError

A `TypeError` is thrown when, for example, trying to run a method on a non-supported data type. A simple example is attempting to run the `pop()` method on a string:

[1](#)

```
"hello".pop() // Uncaught TypeError: "hello".pop is not a function
```

The array-like behavior of strings was already covered in an earlier lesson in this course. However, as can be confirmed by running the above line of code, strings do not have all the array methods readily available to them, and trying to use some of those methods will result in a `TypeError` being thrown.

RangeError

A `RangeError` is thrown when we're giving a value to a function, but that value is out of the allowed range of acceptable input values.

Here's a simple example of converting an everyday *Base 10 number* (a number of the common decimal system) to a *Base 2 number* (i.e binary number).

For example:

[1](#)

```
(10).toString(2); // '1010'
```

The value of `2` when passed to the `toString()` method, is like saying to JavaScript: "convert the value of `10` of the Base 10 number system, to its counter-part in the Base 2 number system". JavaScript obliges and "translates" the "regular" number `10` to its binary counter-part. Besides using Base 2 number system, I can also use the Base 8, like this:

[1](#)

```
(10).toString(8); // 12
```

I get back the value `12`, which is the plain number `10`, written in Base 8 number system. However, if I try to use a non-existing number system, such as an imaginary *Base 100*, since this number system effectively doesn't exist in JavaScript, I will get the `RangeError`, because a non-existing *Base 100* system is **out of range** of the number systems that are available to the `toString()` method:

[1](#)

```
(10).toString(100); // Uncaught RangeError: toString() radix argument must  
be between 2 and 36
```

In this reading, you've covered some of the most common errors in JavaScript.

Completed

Exercise: Error prevention Instructions

Task 1: Code a function declaration

You need to code a function declaration named `addTwoNums`, which accepts numbers `a` and `b` and console logs `a + b`.

Task 2: Invoke the `addTwoNums` function with a number and a string

You need to invoke the `addTwoNums` using the following arguments: 5 and "5".

Task 3: Update the `addTwoNums` function with a `try...catch` block

Add the try and catch blocks inside the function definition's body. For now, just make sure that the console log of `a + b` is inside the try block. Additionally, the `catch` block should catch an error named `err` and, inside the body of the `catch` block, you need to console log the `err` value.

Task 4: If the passed-in arguments are not numbers, throw an error

If either of the arguments passed to the `addTwoNums` are not numbers, you'll throw an error.

Specifically, code a conditional with the following logic:

- if the `typeof` the `a` parameter is not equal to '`number`', throw a new `ReferenceError`. Inside the `ReferenceError`, pass a custom error message of '`the first argument is not a number`'.
- else if the `typeof` the `b` parameter is not equal to '`number`', throw a new `ReferenceError`. Inside the `ReferenceError`, pass a custom error message of '`the second argument is not a number`'.
- else, console log `a + b`

Once you've completed this task, all the code inside the `try` block will be inside these conditional statements.

Task 5: Update the catch block

Inside the catch block, update the code from `console.log(err)` to `console.log("Error!", err)`.

Task 6: Invoke the addTwoNums function

Invoke the `addTwoNums` function using `5` and `"5"` as arguments.

Task 7: Add another console log under the addTwoNums function invocation

Add another line of code that console logs the string `"It still works"`.

1

Run

Reset

Completed

Solution: Error prevention

Solution to task 1: Code a function declaration

1

2

3

```
function addTwoNums(a,b) {  
  
    console.log(a + b) //display the result  
  
}
```

Run

Reset

Solution to task 2: Invoke the addTwoNums function with a number and a string

1

```
function addTwoNums (a,b) {  
  
    console.log(a + b)  
  
}  
  
addTwoNums (5, "5") // "55"
```

Run

Reset

Solution to task 3: Update the `addTwoNums` function with a `try...catch` block

```
function addTwoNums (a,b) {  
  
    try {  
  
        console.log(a + b)  
  
    } catch (err) {
```

2

3

4

1

2

3

4

5

6

7

```
    console.log(err)  
  }  
}
```

Run

Reset

Solution to task 4: If the passed-in arguments are not numbers, throw an error

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13
```

```
function addTwoNums(a,b) {  
  try {  
    if(typeof(a) != 'number') {
```

```
    throw new ReferenceError('the first argument is not a number')

} else if (typeof(b) != 'number') {

    throw new ReferenceError('the second argument is not a
number')

} else {

    console.log(a + b)

}

} catch(err) {

    console.log(err)

}

}
```

[Run](#)

[Reset](#)

Solution to task 5: Update the catch block

1
2
3
4
5
6
7
8

9

10

11

12

13

```
function addTwoNums(a,b) {  
  try {  
    if(typeof(a) != 'number') {  
      throw new ReferenceError('the first argument is not a number')  
    } else if (typeof(b) != 'number') {  
      throw new ReferenceError('the second argument is not a  
number')  
    } else {  
      console.log(a + b)  
    }  
  } catch(err) {  
    console.log("Error!", err)  
  }  
}
```

Run

Reset

Solution to tasks 6 and 7: Invoke the addTwoNums function and Add another console log under the addTwoNums function invocation

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
  
function addTwoNums(a,b) {  
  
    try {  
  
        if(typeof(a) != 'number') {  
  
            throw new ReferenceError('the first argument is not a number')  
  
        } else if (typeof(b) != 'number') {  
  
    }  
}
```

```
        throw new ReferenceError('the second argument is not a
number')

    } else {

        console.log(a + b)

    }

} catch(err) {

    console.log("Error!", err)

}

}

addTwoNums(5, "5")

console.log("It still works")
```

[Run](#)

[Reset](#)

Completed

Exercise: Defensive programming

Defensive programming is all about *assuming* that all the arguments a function will receive are of the wrong type, the wrong value or both.

In other words, you are assuming that things will go wrong and you are proactive in thinking about such scenarios before they happen, so as to make your function less likely to cause errors because of faulty inputs.

How would you then refactor the function given below with defensive programming in mind?

For this exercise, let's make sure that both of the arguments that are passed in satisfy the following criteria:

- The length of the `word` parameter cannot be less than `2`.
- The length of the `match` parameter must be `1`.
- The type of both the `word` and the `match` parameters must be `string`.

You will use the code below to complete your task:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
  
function letterFinder(word, match) {  
  
    for(i = 0; i < word.length; i++) {  
  
        if(word[i] == match) {  
  
            //if the current character at position i in the word is equal  
            to the match  
  
            console.log('Found the', match, 'at', i)  
  
        } else {  
  
            console.log('---No match found at', i)
```

```
    }  
  
}  
  
}
```

[Run](#)

[Reset](#)

Here are the tasks to complete:

1. Just above the `for` loop in the `letterFinder` function definition, declare a variable named `condition1` and assign to it the following code: `typeof(word) == 'string' && word.length >= 2.`
2. Declare a variable named `condition2` on the next line and assign to it and assign to it a check that makes sure that the type of `match` is a string AND that the length of the `match` variable is equal to 1.
3. Write an if statement on the next line that checks that `condition1` is `true`, and `condition2` is `true`
4. Move the rest of the function's body into the if statement you wrote in the previous step.
5. Code an "else" block after the "if" condition and `console.log` the following: `"Please pass correct arguments to the function."`.
6. As a failing test, run the `letterFinder` function and pass it with any two numbers as arguments.
7. As a passing test, run the `letterFinder` funciton and pass it with correct arguments, such as: `letterFinder("cat", "c")`.

1

[Run](#)

[Reset](#)

Completed

Solution: Defensive programming

Answer (full completed code)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

```
function letterFinder(word, match) {  
  
    var condition1 = typeof(word) == 'string' && word.length >= 2; //if  
the word is a string and the length is greater than or equal to 2  
  
    var condition2 = typeof(match) == 'string' && match.length == 1; //if  
the match is a string and the length is equal to 1  
  
    if(condition1 && condition2) { //if both condition matches  
  
        for(var i = 0; i < word.length; i++) {  
  
            if(word[i] == match) {  
  
                //check if the character at this i position in the word is  
equal to the match  
  
                console.log('Found the', match, 'at', i)  
  
            } else {  
  
                console.log('---No match found at', i)  
  
            }  
  
        }  
  
    } else {  
  
        //if the requirements don't match  
  
        console.log("Please pass correct arguments to the function")  
  
    }  
  
}  
  
letterFinder([], [])  
  
letterFinder("cat", "c")
```

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN functions](#)

[MDN try...catch](#)

[Iteration protocols](#)

[The Math object on MDN](#)

[The String object on MDN](#)

[MDN JavaScript error reference](#)

[The null value on MDN](#)

[The undefined property on MDN](#)

Completed

Week3-Return values from functions

Many functions, by default, return the value of `undefined`.

An example is the `console.log()` function.

If I run:

1

```
console.log('Hello');
```

... here's the output in the console:

1

2

```
Hello
```

```
undefined
```

Because the `console.log()` function is built so as to not have the explicitly set return value, it gets the default return value of `undefined`.

I'll now code my own implementation of `console.log()`, which doesn't return the value of `undefined`:

```
1  
2  
3  
4  
  
function consoleLog(val) {  
  
    console.log(val)  
  
    return val  
  
}
```

I'm using the `console.log()` function inside my custom `consoleLog` function declaration. And I'm specifying it to return the value of its argument.

Now when I run my custom `consoleLog()` function:

```
1  
  
consoleLog('Hello')
```

I get the following output:

```
1  
  
2  
  
Hello
```

```
'Hello'
```

So, the value is output in the console, but it's also returned.

Why is this useful?

It's useful because I can use return values from one function inside another function.

Here's an example.

I'll first code a function that returns a double of a number that it received:

```
1  
  
2
```

3

```
function doubleIt(num) {  
  return num * 2  
}
```

Now I'll code another function that builds an object with a specific value:

1

2

3

4

5

```
function objectMaker(val) {  
  return {  
    prop: val  
  }  
}
```

I can call the `objectMaker()` function with any value I like, such as:

1

```
objectMaker(20);
```

The returned value will be an object with a single `prop` key set to 20:

1

```
{prop:20}
```

Now consider this code:

1

```
doubleIt(10).toString()
```

The above code returns the number 20 as a string, that is: "20".

I can even combine my custom function calls as follows:

1

```
objectMaker( doubleIt(100) );
```

This will now return the following value:

1

```
{prop: 200}
```

What does all of this mean?

It means that by JavaScript allowing me to use the `return` keyword as described above, I can have multiple function calls, returning data and manipulating values, based on whatever coding challenge I have in front of me.

Being able to return custom values is one of the foundations that makes functional programming possible.

Completed

The functional programming paradigm

Learning Objectives

- Be able to explain that there are several programming paradigms
- Be able to explain the basic difference between the two predominant programming paradigms: functional programming and object-oriented programming
- Understand, at a very high level, how the functional programming paradigm works

"There are actually several styles of coding, also known as **paradigms**. A common style is called **functional programming**, or FP for short.

In functional programming, we use a lot of functions and variables.

1

2

3

4

5

6

7

```
function getTotal(a,b) {  
    return a + b  
}  
  
var num1 = 2;  
  
var num2 = 3;  
  
var total = getTotal(num1, num2);
```

When writing FP code, we keep data and functionality separate and pass data into functions only when we want something computed.

1

2

3

4

5

6

```
function getDistance(mph, h) {  
    return mph * h  
}  
  
var mph = 60;
```

```
var h = 2;  
  
var distance = getDistance(mph, h);
```

In functional programming, functions return new values and then use those values somewhere else in the code.

1

2

3

4

5

6

7

8

```
function getDistance(mph, h) {  
  
    return mph * h  
  
}  
  
var mph = 60;  
  
var h = 2;  
  
var distance = getDistance(mph, h);  
  
  
console.log(distance); // <===== THIS HERE!
```

Another style is **object-oriented programming (OOP)**. In this style, we group data and functionality as properties and methods inside objects.

For example, if I have a `virtualPet` object, I can give it a `sleepy` property and a `nap()` method:

1

2

3

4

```
var virtualPet = {  
  sleepy: true,  
  nap: function() {}  
}
```

In OOP, methods **update properties** stored in the object instead of generating new return values. For example, if I check the `sleepy` property on the `virtualPet` object, I can confirm that it's set to `true`.

However, once I've ran the `nap()` method on the `virtualPet` object, will the `sleepy` property's value change?

1

2

3

4

5

6

7

8

9

10

```
//creating an object  
  
var virtualPet = {
```

```

sleepy: true,
nap: function() {
    this.sleepy = false
}
}

console.log(virtualPet.sleepy) // true

virtualPet.nap()

console.log(virtualPet.sleepy) // false

```

OOP helps us model real-life objects. It works best when the grouping of properties and data in an object makes logical sense - meaning, the properties and methods "belong together".

Note that the goal here is not to discuss OOP in depth; instead, I just want to show you the simplest explanation of what it is and how it works, in order to make the single most important distinction between FP and OOP.

To summarize this point, we can say that the Functional Programming paradigm works by keeping the data and functionality separate. Its counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects.

There are many more concepts and ideas in functional programming.

Here are some of the most important ones:

- First-class functions
- Higher-order function
- Pure functions and side-effects

There are many other concepts and principles in functional programming, but for now, let's stick to these three.

First-class functions

It is often said that functions in JavaScript are "first-class citizens". What does that mean?

It means that a function in JavaScript is just another value that we can:

- pass to other functions
- save in a variable
- return from other functions

In other words, a function in JavaScript is just a value - from this vantage point, almost no different than a string or a number. For example, in JavaScript, it's perfectly normal to pass a function invocation to another function. To explain how this works, consider the following program.

```
function addTwoNums(a, b) {  
    console.log(a + b)  
}
```

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

```

}

function randomNum() {
    return Math.floor((Math.random() * 10) + 1);
}

function specificNum() { return 42 };

var useRandom = true;

var getNumber;

if(useRandom) {
    getNumber = randomNum
} else {
    getNumber = specificNum
}

addTwoNums(getNumber(), getNumber())

```

I start the program with the `addTwoNums()` function whose definition I've already used earlier in various variations. The reason why this function is a recurring example is because it's so simple that it helps explain concepts that otherwise might be a bit harder to grasp. Next, I code a function named `randomNum()` which returns a random number between 0 and 10. I then code another function named `specificNum()` which returns a specific number, the number 42. Next, I save a variable named `useRandom`, and I set it to the boolean value of `true`. I declare another variable,

named `getNumber`. This is where things get interesting. On the next several lines, I have an if else statement. The if condition is executed when the value of `useRandom` is set to `true`. If that's the case, the entire `randomNum()` function's declaration is saved into the `getNumber` variable. Otherwise, I'm saving the entire `specificNum()` function's declaration into the `getNumber` variable. In other words, based on the `useRandom` being set to `true` or `false`, the `getNumber` variable will be assigned either the `randomNum()` function declaration or the `specificNum()` function declaration. With all this code set, I can then invoke the `addTwoNums()` function, passing it the invocation of the `getNumber()` variables as its first and second arguments. **This works because functions in JavaScript are truly first-class citizens, which can be assigned to variable names and passed around just like I would pass around a string, a number, an object, etc.** Note: most of the code inside the `randomNum()` function declaration comes from a previous lesson, namely the lesson that discussed the Math object in JavaScript. This brings me to the second foundational concept of functional programming, which is the concept of higher-order functions.

Higher-order functions

A higher-order function is a function that has either one or both of the following characteristics:

- It accepts other functions as arguments
- It returns functions when invoked

There's no "special way" of defining higher-order functions in JavaScript. It is simply a feature of the language. The language itself allows me to pass a function to another function, or to return a function from another function. Continuing from the previous section, consider the following code, in which I'm re-defining the `addTwoNums()` function so that it is a higher-order function:

```
1  
2  
3  
  
function addTwoNums (getNumber1, getNumber2) {  
  
    console.log(getNumber1() + getNumber2());  
  
}
```

You can think of the above function declaration of `addTwoNums` as describing how it will deal with the `getNumber1` and `getNumber2` inputs: once it receives them as arguments, it will then attempt invoking them and concatenating the values returned from those invocations. For example:

```
1  
2  
  
addTwoNums (specificNum, specificNum); // returned number is 84
```

```
addTwoNums (specificNum, randomNum); // returned number is 42 + some random  
number
```

Pure functions and side-effects

Another concept of functional programming are pure functions.

A pure function returns the exact same result as long as it's given the same values.

An example of a pure function is the `addTwoNums()` function from the previous section:

```
function addTwoNums (a, b) {  
  
    console.log(a + b)  
  
}
```

This function will always return the same output, based on the input. For example, as long as we give it a specific value, say, a 5, and a 6:

```
addTwoNums (5, 6); // 11
```

... the output will always be the same.

Another rule for a function to be considered pure is that it should not have side-effects. A side-effect is any instance where a function makes a change outside of itself.

This includes:

- changing variable values outside of the function itself, or even relying on outside variables
- calling a Browser API (even the `console` itself!)
- calling `Math.random()` - since the value cannot be reliably repeated

The topic of pure and impure functions can get somewhat complex.

For now, it's sufficient to know that this concept exists and that it is related to functional programming.

Completed

Visual Studio Code on Coursera

In addition to having Visual Studio Code installed on your own computer, in this course and throughout this program, you'll have the opportunity to work in Visual Studio Code right here on Coursera!

As you progress through the course, you'll be able to write code in hands-on activities called **Labs**. In these labs you'll be able to open Visual Studio Code and start writing code without ever leaving the course.

You'll have plenty of opportunities to see Labs in action later in the course, but for now, use the images below as a visual guide to how Labs will look and operate in your browser.

Lab: Creating an HTML Document

The Labs contain instructions explaining the coding task.

Creating an HTML Document

[Open Lab ↗](#)

Instructions



Introduction

In this ungraded lab you will practice creating a simple HTML document.

Goal

- Create a valid HTML document that displays a piece of text.

Objectives

- Add the DOCTYPE.
- Add the HTML, head and body elements.
- Add the title element.
- Add the text to the body element.

Instructions

When you click the button to open the lab, a new tab will open with Visual Studio Code already setup and ready for you to start writing code!



☰ Navigate ⚒ Lab Files ⓘ Help

The screenshot shows the Visual Studio Code interface. The top bar includes the Coursera logo, navigation links (☰ Navigate, ⚒ Lab Files, ⓘ Help), and file tabs (index.html — project — code-server). The left sidebar (Explorer) shows a project structure with a folder named ".dotfiles-coursera" containing an "index.html" file. The main editor area displays the content of "index.html". The bottom status bar shows connection details (wdblkcv.a.labs.coursera.org), code analysis results (0△0), and various development settings like Open Development Server, Layout: U.S., ESLint, and Prettier.

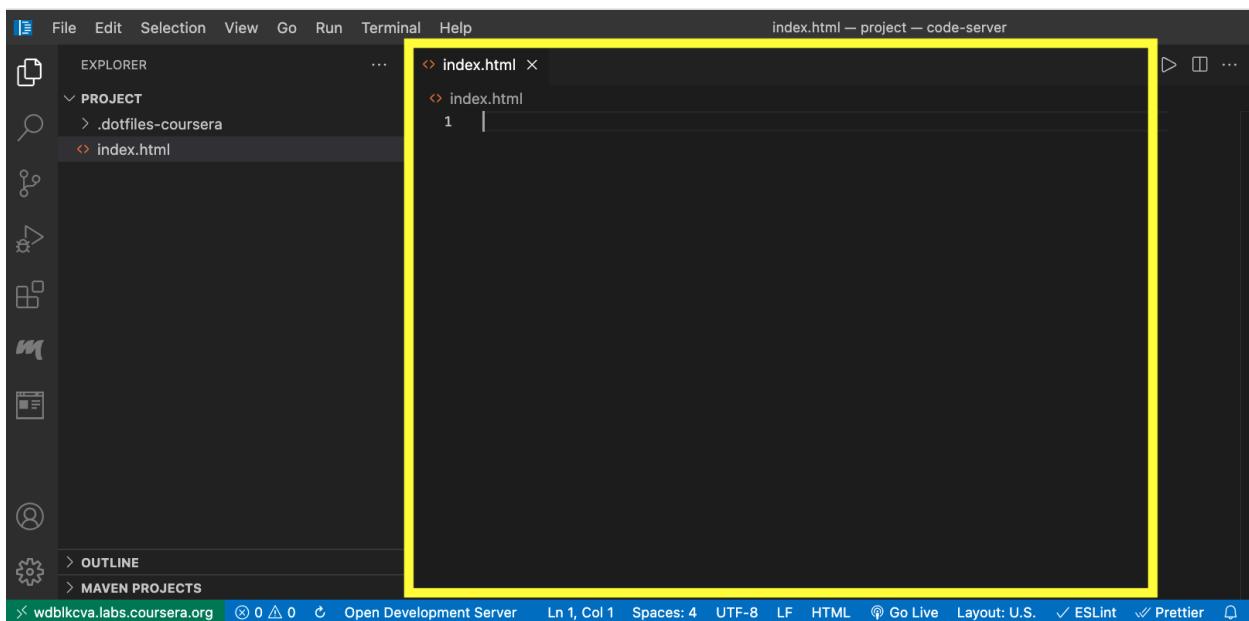
You'll see all the files for the lab in the Project folder in the left sidebar.



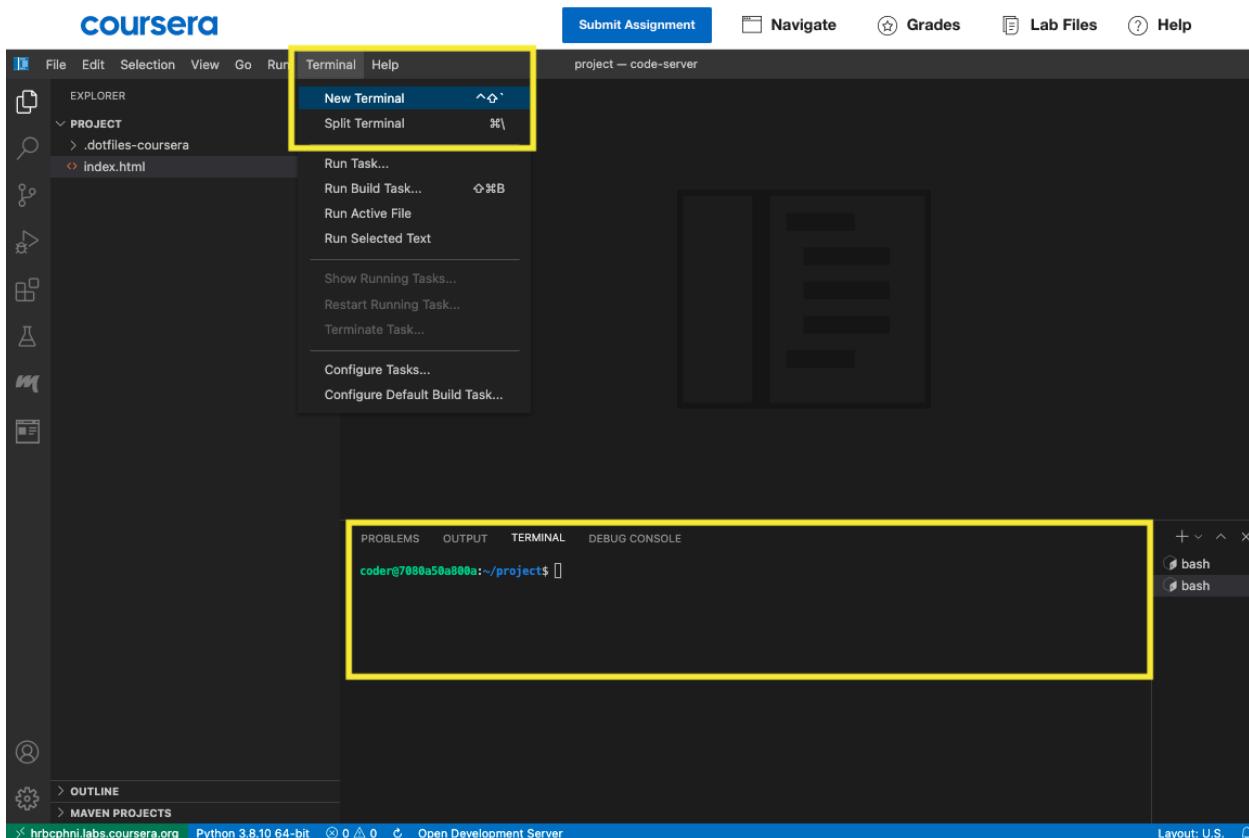
☰ Navigate ⚒ Lab Files ⓘ Help

This screenshot is identical to the one above, but the Explorer sidebar on the left is highlighted with a thick yellow border. The rest of the interface, including the editor area and the bottom status bar, remains the same.

And the large editor area where you write your code for the lab.

[Navigate](#)[Lab Files](#)[Help](#)

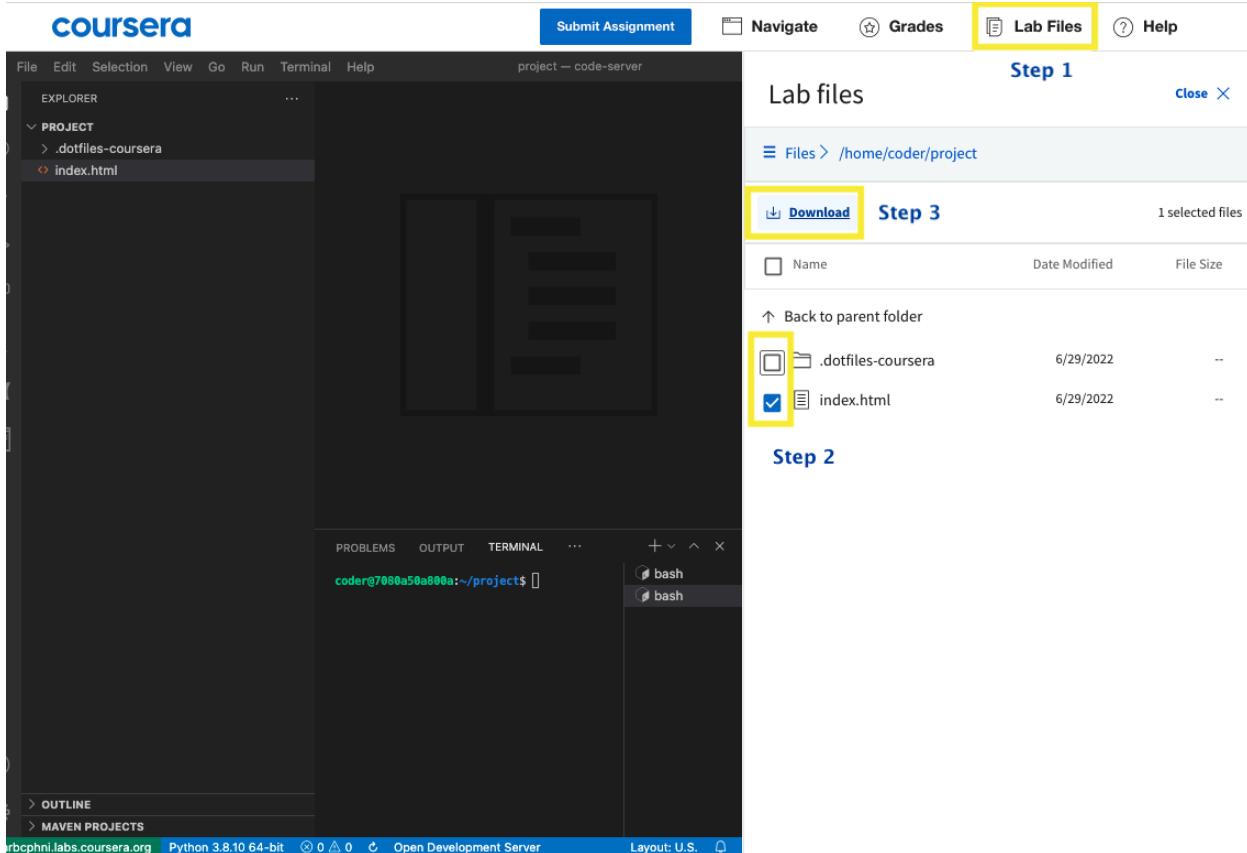
You may need to use a tool called the Terminal from time to time to complete course activities. You can open this by selecting the **Terminal** option in the upper Visual Studio Code toolbar.



How to download files from your Visual Studio Code Lab to your local device

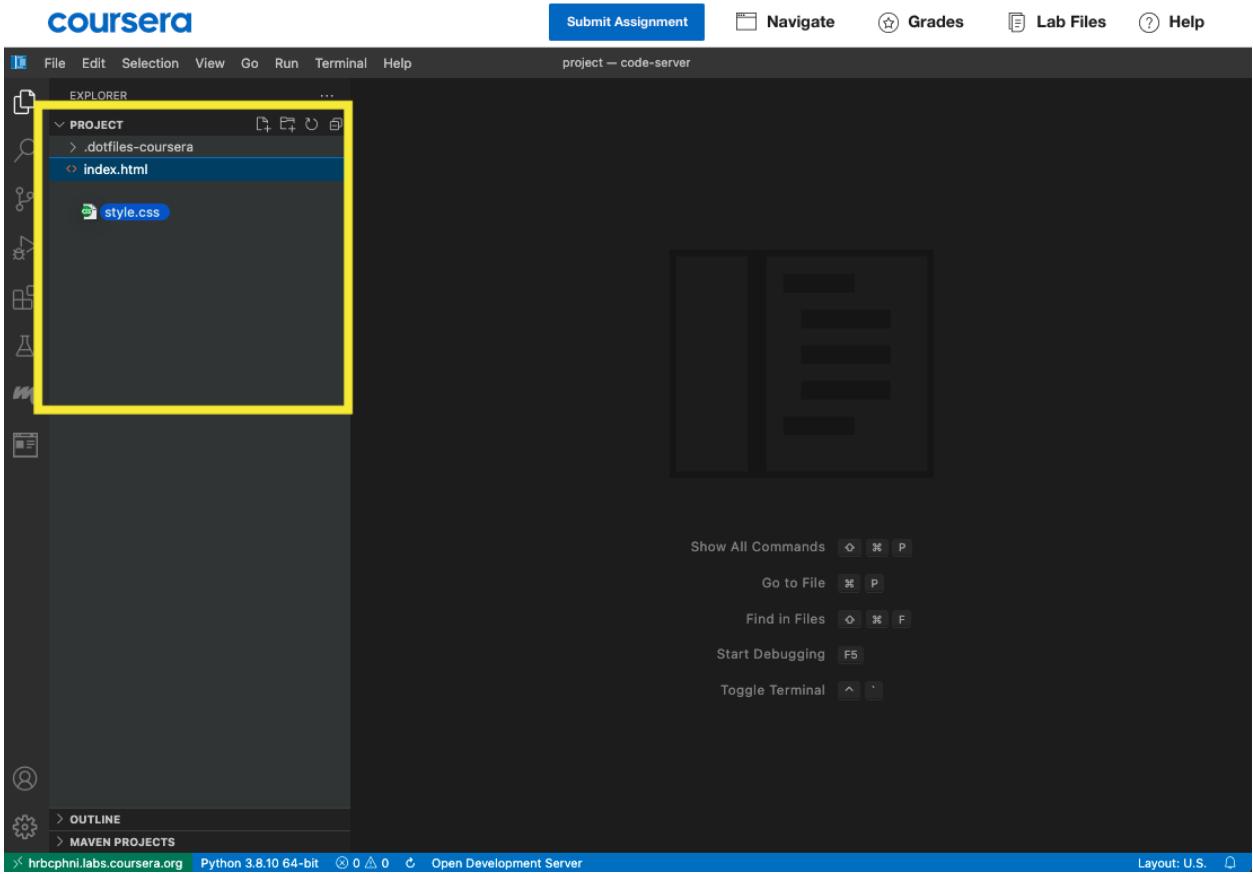
1. Select the **Lab Files** button in your Lab Toolbar.

2. You'll be able to download your full workspace, specific folders, or individual files through the checkbox selection tool.
3. After you've selected these files, use the **Download** link to download your files to your local device.



How to upload local files to your Visual Studio Code Lab

If you'd like to upload your course files from your local device to your Visual Studio Code lab, **drag and drop** your file from your local device into the Visual Studio Code file tree.



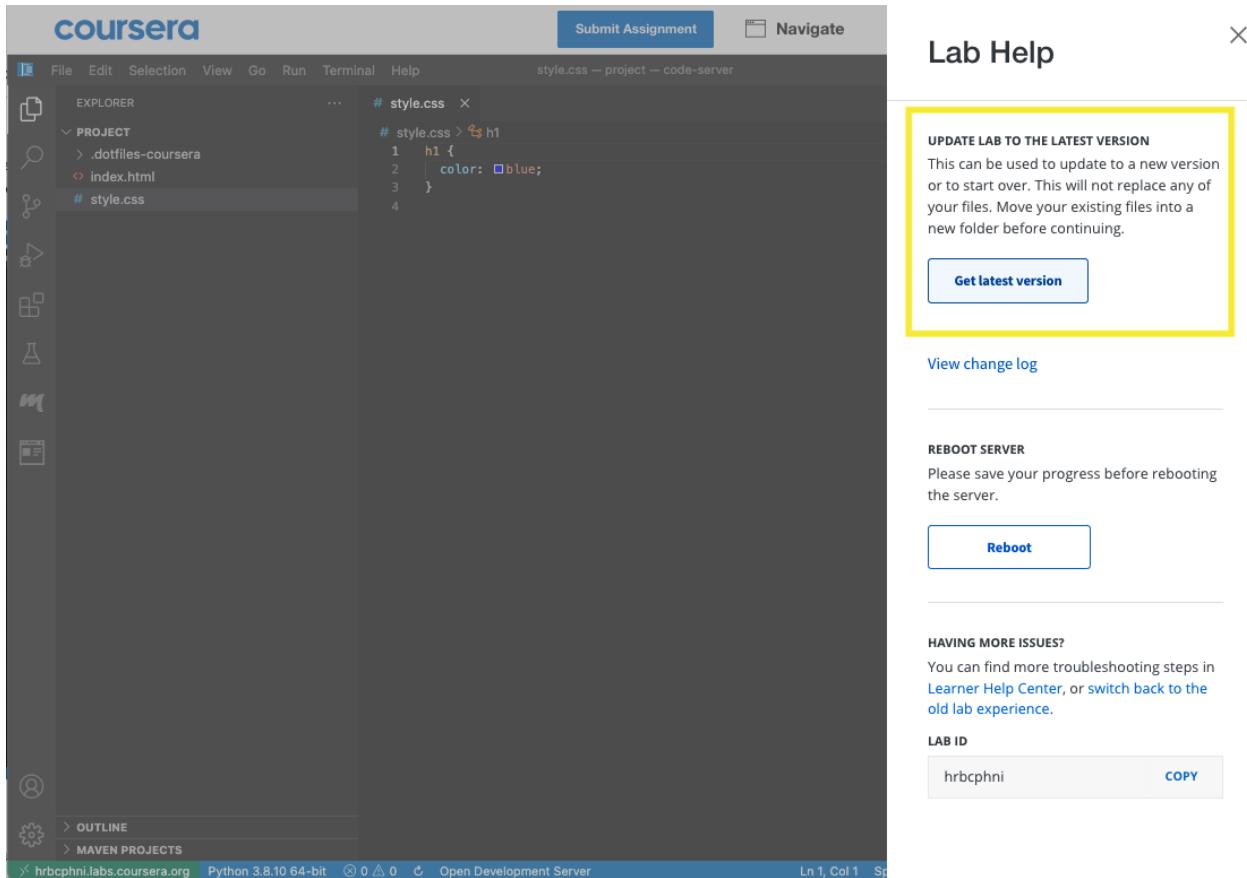
How to get a fresh copy of course-provided starter files

Your work will be saved and persist within your Visual Studio Code lab while you are enrolled in the course. If you'd like to get a fresh copy of the original instructor-provided files at any time, you can do this through the **Lab Help** option in your Lab Toolbar. Don't worry - your original work and files will still remain in your lab until you personally remove or delete them, even when refreshing your files through the steps below.

1. First rename your original files to something like `[yourfilename] [original].[your file extension]`. You can do this by right-clicking on your file in the Visual Studio Code file tree, selecting **Rename**, and providing a new file name.

- For example for `index.html`, this could be renamed to `'index [original].html'`

2. Select **Lab Help** from your Lab Toolbar and then select **Get latest version**.



3. You should now see a fresh copy of the original instructor-provided files in your lab, in addition to your own (renamed) files.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN Functions Guide](#)

[MDN Glossary: Recursion](#)

[MDN Glossary: Scope](#)

[Functional Programming in JavaScript](#)

[MDN: First-class functions](#)

Completed

Object Oriented Programming principles

In this reading, you'll learn about the benefits of object-oriented programming (OOP) and the OOP principles.

The Benefits of OOP

There are many benefits to using the object-oriented programming (OOP) paradigm.

OOP helps developers to mimic the relationship between objects in the real world. In a way, it helps you to reason about relationships between things in your software, just like you would in the real world. Thus, OOP is an effective approach to come up with solutions in the code you write. OOP also:

- Allows you to write modular code,
- Makes your code more flexible and
- Makes your code reusable.

The Principles of OOP

The four fundamental OOP principles are inheritance, encapsulation, abstraction and polymorphism. You'll learn about each of these principles in turn. The thing to remember about Objects is that they exist in a hierachal structure. Meaning that the original base or super class for everything is the Object class, all objects derive from this class. This allows us to utilize the Object.create() method. to create or instansiate objects of our classes.

1

2

3

4

5

```
class Animal { /* ...class code here... */ }
```

```
var myDog = Object.create(Animal)
```

```
console.log (Animal)
```

[Run](#)

[Reset](#)

A more common method of creating objects from classes is to use the `new` keyword. When using a default or empty constructor method, JavaScript implicitly calls the Object superclass to create the instance.

1

2

3

4

5

```
class Animal { /* ...class code here... */ }
```

```
var myDog = new Animal()
```

```
console.log (Animal)
```

[Run](#)

[Reset](#)

This concept is explored within the next section on inheritance

OOP Principles: Inheritance

Inheritance is one of the foundations of object-oriented programming.

In essence, it's a very simple concept. It works like this:

1. There is a base class of a "thing".
2. There is one or more sub-classes of "things" that inherit the properties of the base class (sometimes also referred to as the "super-class")
3. There might be some other sub-sub-classes of "things" that inherit from those classes in point 2.

Note that each sub-class inherits from its super-class. In turn, a sub-class might also be a super-class, if there are classes inheriting from that sub-class.

All of this might sound a bit "computer-sciency", so here's a more practical example:

1. There is a base class of "Animal".
2. There is another class, a sub-class inheriting from "Animal", and the name of this class is "Bird".
3. Next, there is another class, inheriting from "Bird", and this class is "Eagle".

Thus, in the above example, I'm modelling objects from the real world by constructing relationships between Animal, Bird, and Eagle. Each of them are separate classes, meaning, each of them are separate blueprints for specific object instances that can be constructed as needed.

To setup the inheritance relation between classes in JavaScript, I can use the `extends` keyword, as in `class B extends A`.

Here's an example of an inheritance hierarchy in JavaScript:

```
1
2
3

class Animal { /* ...class code here... */ }

class Bird extends Animal { /* ...class code here... */ }

class Eagle extends Bird { /* ...class code here... */ }
```

OOP Principles: Encapsulation

In the simplest terms, encapsulation has to do with making a code implementation "hidden" from other users, in the sense that they don't have to know how my code works in order to "consume" the code.

For example, when I run the following code:

```
1

"abc".toUpperCase();
```

I don't really need to worry or even waste time thinking about how the `toUpperCase()` method works. All I want is to use it, since I know it's available to me. Even if the underlying syntax - that is, the implementation of the `toUpperCase()` method changes - as long as it doesn't break my code, I don't have to worry about what it does in the background, or even how it does it.

OOP Principles: Abstraction

Abstraction is all about writing code in a way that will make it more generalized.

The concepts of encapsulation and abstraction are often misunderstood because their differences can feel blurry.

It helps to think of it in the following terms:

- An abstraction is about extracting the *concept* of what you're trying to do, rather than dealing with a specific manifestation of that concept.

- Encapsulation is about you not having access to, or not being concerned with, how some implementation works internally.

While both the encapsulation and abstraction are important concepts in OOP, it requires more experience with programming in general to really delve into this topic.

For now, it's enough to be aware of their existence in OOP.

OOP Principles: Polymorphism

Polymorphism is a word derived from the Greek language meaning "multiple forms". An alternative translation might be: "something that can take on many shapes".

So, to understand what polymorphism is about, let's consider some real-life objects.

- A door has a bell. It could be said that the bell is a property of the door object. This bell can be rung. When would someone ring a bell on the door? Obviously, to get someone to show up at the door.
- Now consider a bell on a bicycle. A bicycle has a bell. It could be said that the bell is a property of the bicycle object. This bell could also be rung. However, the reason, the intention, and the result of somebody ringing the bell on a bicycle is not the same as ringing the bell on a door.

The above concepts can be coded in JavaScript as follows:

```
1
2
3
4
5
6
7
8
9
10
const bicycle = {
  bell: function() {
    console.log('Ringing the bell on a bicycle');
  }
};
```

```
        return "Ring, ring! Watch out, please!"  
    }  
  
}  
  
const door = {  
  
    bell: function() {  
  
        return "Ring, ring! Come here, please!"  
    }  
  
}
```

So, I can access the `bell()` method on the `bicycle` object, using the following syntax:

[1](#)

```
bicycle.bell(); // "Get away, please"
```

I can also access the `bell()` method on the `door` object, using this syntax:

[1](#)

```
door.bell(); // "Come here, please"
```

At this point, one can conclude that the exact same name of the method can have the exact opposite intent, based on what object it is used for.

Now, to make this code truly polymorphic, I will add another function declaration:

[1](#)

[2](#)

[3](#)

```
function ringTheBell(thing) {  
  
    console.log(thing.bell())  
  
}
```

Now I have declared a `ringTheBell()` function. It accepts a `thing` parameter - which I expect to be an object, namely, either the `bicycle` object or the `door` object.

So now, if I call the `ringTheBell()` function and pass it the `bicycle` as its single argument, here's the output:

1

```
ringTheBell(bicycle); // Ring, ring! Watch out, please!
```

However, if I invoke the `ringTheBell()` function and pass it the `door` object, I'll get the following output:

1

```
ringTheBell(door); // "Ring, ring! Come here, please!"
```

You've now seen an example of the exact same function producing different results, **based on the context** in which it is used.

Here's another example, the concatenation operator, used by calling the built-in `concat()` method. If I use the `concat()` method on two strings, it behaves exactly the same as if I used the `+` operator.

1

```
"abc".concat("def"); // 'abcdef'
```

I can also use the `concat()` method on two arrays. Here's the result:

1

```
["abc"].concat(["def"]); // ['abc', 'def']
```

Consider using the `+` operator on two arrays with one member each:

1

2

```
["abc"] + ["def"]; // ["abcdef"]
```

This means that the `concat()` method is exhibiting polymorphic behavior since it behaves differently based on the context - in this case, based on what data types I give it.

To reiterate, polymorphism is useful because it allows developers to build objects that can have the exact same functionality, namely, functions with the exact same name, which behave exactly the same. However, at the same time, you can override some parts of the shared functionality or even the complete functionality, in some other parts of the OOP structure.

Here's an example of polymorphism using classes in JavaScript:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

```
class Bird {  
  
    useWings() {  
  
        console.log("Flying!")  
  
    }  
  
}  
  
class Eagle extends Bird {  
  
    useWings() {  
  
        super.useWings()  
  
        console.log("Barely flapping!")  
  
    }  
  
}  
  
class Penguin extends Bird {  
  
    useWings() {  
  
        console.log("Diving!")  
  
    }  
  
}  
  
var baldEagle = new Eagle();  
  
var kingPenguin = new Penguin();  
  
baldEagle.useWings(); // "Flying! Barely flapping!"  
  
kingPenguin.useWings(); // "Diving!"
```

The `Penguin` and `Eagle` sub-classes both inherit from the `Bird` super-class. The `Eagle` sub-class inherits the `useWings()` method from the `Bird` class, but extends it with an additional console log. The `Penguin` sub-class doesn't inherit the `useWings()` class - instead, it has its own implementation, although the `Penguin` class itself does extend the `Bird` class. Do some practice with the above code, try creating some of your own classes. (hint : think about things you know from everyday life)

1

```
// create your classes here
```

Run

Reset

Completed

Constructors

JavaScript has a number of built-in object types, such as:

`Math`, `Date`, `Object`, `Function`, `Boolean`, `Symbol`, `Array`, `Map`, `Set`, `Promise`, `JSON`, etc.

These objects are sometimes referred to as "native objects".

Constructor functions, commonly referred to as just "constructors", are special functions that allow us to build instances of these built-in native objects. All the constructors are capitalized.

To use a constructor function, I must prepend it with the operator `new`.

For example, to create a new instance of the `Date` object, I can run: `new Date()`. What I get back is the current datetime, such as:

```
Thu Feb 03 2022 11:24:08 GMT+0100 (Central European Standard Time)
```

However, not all the built-in objects come with a constructor function. An example of such an object type is the built-in `Math` object.

Running `new Math()` throws an `Uncaught TypeError`, informing us that `Math` is not a constructor.

Thus, I can conclude that some built-in objects do have constructors, when they serve a particular purpose: to allow us to instantiate a specific instance of a given object's constructor. The built-in `Date` object is perfectly suited for having a constructor because each new date object instance I build should have unique data by definition, since it's going to be a different timestamp - it's going to be built at a different moment in time.

Other built-in objects that don't have constructors, such as the `Math` object, don't need a constructor. They're just static objects whose properties and methods can be accessed directly, from the built-in object itself. In other words, there is no point in building an instance of the built-in `Math` object to be able to use its functionality.

For example, if I want to use the `pow` method of the `Math` object to calculate exponential values, there's no need to build an instance of the `Math` object to do so. For example, to get the number 2 to the power of 5, I'd run:

```
Math.pow(2, 5); // --> 32
```

There's no need to build an instance of the `Math` object since there would be nothing that needs to be stored in that specific object's instance.

Besides constructor functions for the built-in objects, I can also define custom constructor functions. Here's an example:

```
1  
2  
3  
4  
5  
6  
  
function Icecream(flavor) {  
  
    this.flavor = flavor;  
  
    this.meltIt = function() {  
  
        console.log(`The ${this.flavor} icecream has melted`);  
  
    }  
  
}  
  
Run  
  
Reset
```

Now I can make as many icecreams as I want:

```
1  
2  
3
```

```
4
5
6
7
8
9
10
11

function Icecream(flavor) {
    this.flavor = flavor;
    this.meltIt = function() {
        console.log(`The ${this.flavor} icecream has melted`);
    }
}

let kiwiIcecream = new Icecream("kiwi");
let appleIcecream = new Icecream("apple");
kiwiIcecream; // --> Icecream {flavor: 'kiwi', meltIt: f}
appleIcecream; // --> Icecream {flavor: 'apple', meltIt: f}
```

Run

Reset

I've just built two instance objects of `Icecream` type.

The most common use case of `new` is to use it with one of the built-in object types.

Note that using constructor functions on all built-in objects is sometimes not the best approach.

This is especially true for object constructors of primitive types, namely: `String`, `Number`, and `Boolean`.

For example, using the built-in `String` constructor, I can build new strings:

1

2

```
let apple = new String("apple");  
  
apple; // --> String {'apple'}
```

The `apple` variable is an object of type `String`.

Let's see how the `apple` object differs from the following `pear` variable:

1

2

```
let pear = "pear";  
  
pear; // --> "pear"
```

The `pear` variable is a string literal, that is, a primitive Javascript value.

The `pear` variable, being a primitive value, will always be more performant than the `apple` variable, which is an object.

Besides being more performant, due to the fact that each object in JavaScript is unique, you can't compare a String object with another String object, even when their values are identical.

In other words, if you compare `new String('plum') === new String('plum')`, you'll get back `false`, while `"plum" === "plum"` returns true. You're getting the `false` when comparing objects because it is not the values that you pass to the constructor that are being compared, but rather the memory location where objects are saved.

Besides not using constructors to build object versions of primitives, you are better off not using constructors when constructing plain, regular objects.

Instead of `new Object`, you should stick to the object literal syntax: `{ }`.

A RegExp object is another built-in object in JavaScript. It's used to **pattern-match strings** using what's known as "Regular Expressions". Regular Expressions exist in many languages, not just JavaScript.

In JavaScript, you can built an instance of the RegExp constructor using `new RegExp`.

Alternatively, you can use a pattern literal instead of RegExp. Here's an example of using `/d/` as a pattern literal, passed-in as an argument to the `match` method on a string.

1

```
"abcd".match(/d/); // null

"abcd".match(/a/); // ['a', index: 0, input: 'abcd', groups: undefined]
```

Instead of using `Array`, `Function`, and `RegExp` constructors, you should use their array literal, function literal, and pattern literal varieties: `[]`, `()`, `{}`, and `/()/.`

However, when building objects of other built-in types, we can use the constructor.

Here are a few examples:

```
1
2
3
4
5
6
7
```

```
new Date();
new Error();
new Map();
new Promise();
new Set();
new WeakSet();
new WeakMap();
```

The above list is inconclusive, but it's just there to give you an idea of some constructor functions you can surely use.

Note that there are links provided about `RegExp` and regular expression in the lesson item titled "*Additional Reading*".

Completed

Creating classes

By the end of this reading, you should be able to explain, with examples, the concept of extending classes using basic inheritance to alter behaviors within child classes.

By now, you should know that inheritance in JavaScript is based around the prototype object.

All objects that are built from the prototype share the same functionality.

When you need to code more complex OOP relationships, you can use the `class` keyword and its easy-to-understand and easy-to-reason-about syntax.

Imagine that you need to code a `Train` class.

Once you've coded this class, you'll be able to use the keyword `new` to instantiate objects of the `Train` class.

For now though, you first need to define the `Train` class, using the following syntax:

1

```
class Train {}
```

So, you use the `class` keyword, then specify the name of your class, with the first letter capitalized, and then you add an opening and a closing curly brace.

In between the curly braces, the first piece of code that you need to define is the `constructor`:

1

2

3

4

5

```
class Train {  
  
    constructor() {
```

```
    }  
  
}
```

The `constructor` will be used to build properties on the future object instance of the `Train` class. For now, let's say that there are only two properties that each object instance of the `Train` class should have at the time it gets instantiated: `color` and `lightsOn`.

```
1  
2  
3  
4  
5  
6  
  
class Train {  
  
  constructor(color, lightsOn) {  
  
    this.color = color;  
  
    this.lightsOn = lightsOn;  
  
  }  
  
}
```

Notice the syntax of the constructor. The constructor is a special function in my `Train` class. First of all, notice that there is no `function` keyword. Also, notice that the keyword `constructor` is used to define this function. You give your `constructor` function parameters inside an opening and closing parenthesis, just like in regular functions. The names of parameters are `color` and `lightsOn`.

Next, inside the `constructor` function's body, you assigned the passed-in `color` parameter's value to `this.color`, and the passed-in `lightsOn` parameter's value to `this.lightsOn`.

What does this `this` keyword here represent?

It's the future object instance of the Train class.

Essentially, this is all the code that you need to write to achieve two things:

1. This code allows me to **build new instances of the Train class**.

- Each object instance of the `Train` class that I build will have its own custom properties of `color` and `lightsOn`.

Now, to actually build a new instance of the `Train` class, I need to use the following syntax:

1

```
new Train()
```

Inside the parentheses, you need to pass values such as `"red"` and `false`, for example, meaning that the `color` property is set to `"red"` and the `lightsOn` property is set to `false`.

And, to be able to interact with the new object built this way, you need to assign it to a variable.

Putting it all together, here's your first train:

1

```
var myFirstTrain = new Train('red', false);
```

Just like any other variable, you can now, for example, console log the `myFirstTrain` object:

1

```
console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}
```

You can continue building instances of the `Train` class. Even if you give them exactly the same properties, they are still separate objects.

1

2

```
var mySecondTrain = new Train('blue', false);
```

```
var myThirdTrain = new Train('blue', false);
```

However, this is not all that classes can offer.

You can also add methods to classes, and these methods will then be shared by all future instance objects of my `Train` class.

For example:

1

2

3

4

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

class Train {
    constructor(color, lightsOn) {
        this.color = color;
        this.lightsOn = lightsOn;
    }
    toggleLights() {

```

```

    this.lightsOn = !this.lightsOn;

}

lightsStatus() {
    console.log('Lights on?', this.lightsOn);

}

getSelf() {
    console.log(this);

}

getPrototypeOf() {
    var proto = Object.getPrototypeOf(this);

    console.log(proto);

}
}

```

Now, there are four methods on your `Train` class: `toggleLights()`, `lightsStatus()`, `getSelf()` and `getPrototypeOf()`.

1. The `toggleLights` method uses the logical not operator, `!`. This operator will change the value stored in the `lightsOn` property of the future instance object of the `Train` class; hence the `!this.lightsOn`. And the `=` operator to its left means that it will get assigned to `this.lightsOn`, meaning that it will become the new value of the `lightsOn` property on that given instance object.
2. The `lightsStatus()` method on the `Train` class just reports the current status of the `lightsOn` variable of a given object instance.
3. The `getSelf()` method prints out the properties on the object instance it is called on.
4. The `getPrototypeOf()` console logs the prototype of the object instance of the `Train` class. The prototype holds all the properties shared by all the object instances of the `Train` class. To get the prototype, you'll be using JavaScript's built-in `Object.getPrototypeOf()` method, and passing it `this` object - meaning, the object instance inside of which this method is invoked.

Now you can build a brand new train using this updated `Train` class:

1

```
var train4 = new Train('red', false);
```

And now, you can run each of its methods, one after the other, to confirm their behavior:

1

2

3

4

```
train4.toggleLights(); // undefined  
  
train4.lightsStatus(); // Lights on? true  
  
train4.getSelf(); // Train {color: 'red', lightsOn: true}  
  
train4.getPrototypeOf(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototypeOf: f}
```

The result of calling `toggleLights()` is the change of true to false and vice-versa, for the `lightsOn` property.

The result of calling `lightsStatus()` is the console logging of the value of the `lightsOn` property. The result of calling `getSelf()` is the console logging the entire object instance in which the `getSelf()` method is called. In this case, the returned object is the `train4` object. Notice that this object gets returned only with the properties ("data") that was build using the `constructor()` function of the `Train` class. That's because all the methods on the `Train` class do not "live" on any of the instance objects of the `Train` class - instead, they live on the prototype, as will be confirmed in the next paragraph.

Finally, the result of calling the `getPrototypeOf()` method is the console logging of all the properties on the `prototype`. When the `class` syntax is used in JavaScript, this results in **only shared methods being stored on the prototype**, while the `constructor()` function sets up the mechanism for saving instance-specific values ("data") at the time of object instantiation.

Thus, in conclusion, the class syntax in JavaScript allows us to clearly separate individual object's data - which exists on the object instance itself - from the shared object's functionality (methods), which exist on the prototype and are shared by all object instances.

However, this is not the whole story.

It is possible to implement polymorphism using classes in JavaScript, by inheriting from the base class and then overriding the inherited behavior. To understand how this works, it is best to use an example.

In the code that follows, you will observe another class being coded, which is named `HighSpeedTrain` and inherits from the `Train` class.

This makes the `Train` class a base class, or the super-class of the `HighSpeedTrain` class. Put differently, the `HighSpeedTrain` class becomes the sub-class of the `Train` class, because it inherits from it.

To inherit from one class to a new sub-class, JavaScript provides the `extends` keyword, which works as follows:

```
1  
2  
  
class HighSpeedTrain extends Train {  
  
}
```

As in the example above, the sub-class syntax is consistent with how the base class is defined in JavaScript. The only addition here is the `extends` keyword, and the name of the class from which the sub-class inherits.

Now you can describe how the `HighSpeedTrain` works. Again, you can start by defining its constructor function:

```
1  
2  
3  
4  
5  
6  
7  
  
class HighSpeedTrain extends Train {  
  
    constructor(passengers, highSpeedOn, color, lightsOn) {  
  
        super(color, lightsOn);  
  
        this.passengers = passengers;  
    }  
}
```

```
class HighSpeedTrain extends Train {  
  
    constructor(passengers, highSpeedOn, color, lightsOn) {  
  
        super(color, lightsOn);  
  
        this.passengers = passengers;  
    }  
}
```

```
    this.highSpeedOn = highSpeedOn;

}

}
```

Notice the slight difference in syntax in the constructor of the `HighSpeedTrain` class, namely the use of the `super` keyword.

In JavaScript classes, `super` is used to specify what property gets inherited from the super-class in the sub-class.

In this case, I choose to inherit both the properties from the `Train` super-class in the `HighSpeedTrain` sub-class.

These properties are `color` and `lightsOn`.

Next, you add the additional properties of the `HighSpeedTrain` class inside its constructor, namely, the `passengers` and `highSpeedOn` properties.

Next, inside the constructor body, you use the `super` keyword and pass in the inherited `color` and `lightsOn` properties that come from the `Train` class. On subsequent lines you assign `passengers` to `this.passengers`, and `highSpeedOn` to `this.highSpeedOn`.

Notice that in addition to the inherited properties, you also **automatically inherit** all the methods that exist on the `Train` prototype, namely, the `toggleLights()`, `lightsStatus()`, `getSelf()`, and `getPrototypeOf()` methods.

Now let's add another method that will be specific to the `HighSpeedTrain` class: the `toggleHighSpeed()` method.

1

2

3

4

5

6

7

8

9

10

11

```
class HighSpeedTrain extends Train {  
  
    constructor(passengers, highSpeedOn, color, lightsOn) {  
  
        super(color, lightsOn);  
  
        this.passengers = passengers;  
  
        this.highSpeedOn = highSpeedOn;  
  
    }  
  
    toggleHighSpeed() {  
  
        this.highSpeedOn = !this.highSpeedOn;  
  
        console.log('High speed status:', this.highSpeedOn);  
  
    }  
  
}
```

Additionally, imagine you realized that you don't like how the `toggleLights()` method from the super-class works, and you want to implement it a bit differently in the sub-class. You can add it inside the `HighSpeedTrain` class.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

```
class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLigths();

        super.lightsStatus();

    }

}
```

```
        console.log('Lights are 100% operational.');
```

```
}
```

```
}
```

So, how did you override the behavior of the original `toggleLights()` method?
Well in the super-class, the `toggleLights()` method was defined as follows:

1

2

3

```
toggleLights() {
```

```
    this.lightsOn = !this.lightsOn;
```

```
}
```

You realized that the `HighSpeedTrain` method should reuse the existing behavior of the original `toggleLights()` method, and so you used the `super.toggleLights()` syntax to inherit the entire super-class' method.

Next, you also inherit the behavior of the super-class' `lightsStatus()` method - because you realize that you want to have the updated status of the `lightsOn` property logged to the console, whenever you invoke the `toggleLights()` method in the sub-class.

Finally, you also add the third line in the re-implemented `toggleLights()` method, namely:

1

```
console.log('Lights are 100% operational.');
```

You've added this third line to show that I can combine the "borrowed" method code from the super-class with your own custom code in the sub-class.

Now you're ready to build some train objects.

1

2

```
var train5 = new Train('blue', false);
```

```
var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);
```

You've built the `train5` object of the `Train` class, and set its `color` to "blue" and its `lightsOn` to `false`.

Next, you've built the `highSpeed1` object to the `HighSpeedTrain` class, setting `passengers` to 200, `highSpeedOn` to `false`, `color` to "green", and `lightsOn` to false.

Now you can test the behavior of `train5`, by calling, for example, the `toggleLights()` method, then the `lightsStatus()` method:

1
2

```
train5.toggleLights(); // undefined  
  
train5.lightsStatus(); // Lights on? true
```

Here's the entire completed code for this lesson:

1
2
3
4
5
6
7
8
9
10
11
12
13
14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
class Train {  
  
    constructor(color, lightsOn) {  
  
        this.color = color;  
  
        this.lightsOn = lightsOn;  
  
    }  
  
    toggleLights() {  
  
        this.lightsOn = !this.lightsOn;  
  
    }  
  
    lightsStatus() {  
  
        console.log('Lights on?', this.lightsOn);  
  
    }  
  
    getSelf() {  
  
        console.log(this);  
  
    }  
  
    getPrototypeOf() {  
  
        var proto = Object.getPrototypeOf(this);  
  
    }  
}
```

```
        console.log(proto);

    }

}

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLights();

        super.lightsStatus();

        console.log('Lights are 100% operational.');

    }

}
```

```
var myFirstTrain = new Train('red', false);

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

var mySecondTrain = new Train('blue', false);
```

Notice how the `toggleLights()` method behaves differently on the `HighSpeedTrain` class than it does on the `Train` class.

Additionally, it helps to visualize what is happening by getting the prototype of both the `train5` and the `highSpeed1` trains:

```
1

2

train5.getPrototypeOf(); // {constructor: f, toggleLights: f, lightsStatus: f, getSelf: f, getPrototypeOf: f}

highSpeed1.getPrototypeOf(); // Train {constructor: f, toggleHighSpeed: f, toggleLights: f}
```

The returned values in this case might initially seem a bit tricky to comprehend, but actually, it is quite simple:

1. The prototype object of the `train5` object was created when you defined the class `Train`. You can access the prototype using `Train.prototype` syntax and get the prototype object back.
2. The prototype object of the `highSpeed1` object is this object: `{constructor: f, toggleHighSpeed: f, toggleLights: f}`. In turn this object has its own prototype, which can be found using the following syntax: `HighSpeedTrain.prototype.__proto__`. Running this code returns: `{constructor: f, toggleLights: f, lightsStatus: f, getSelf: f, getPrototypeOf: f}`.

Prototypes seem easy to grasp at a certain level, but it's easy to get lost in the complexity. This is one of the reasons why class syntax in JavaScript improves your developer experience, by making it easier to reason about the relationships between classes. However, as you improve your skills, you should always strive to understand your tools better, and this includes prototypes. After all, JavaScript is just a tool, and you've now "peeked behind the curtain".

In this reading, you've learned the very essence of how OOP with classes works in JavaScript. However, this is not all.

In the lesson on designing an object-oriented program, you'll learn some more useful concepts. These mostly have to do with coding your classes so that it's even easier to create object instances of those classes in JavaScript.

Using class instance as another class' constructor's property

Consider the following example:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

```
class StationaryBike {  
  
    constructor(position, gears) {  
  
        this.position = position  
  
        this.gears = gears  
  
    }  
  
}
```

```
class Treadmill {  
  
    constructor(position, modes) {  
  
        this.position = position  
  
        this.modes = modes  
  
    }  
  
}
```

```
class Gym {  
  
    constructor(openHrs, stationaryBikePos, treadmillPos) {
```

```

        this.openHrs = openHrs

        this.stationaryBike = new StationaryBike(stationaryBikePos, 8)

        this.treadmill = new Treadmill(treadmillPos, 5)

    }

}

var boxingGym = new Gym("7-22", "right corner", "left corner")

console.log(boxingGym.openHrs) //

console.log(boxingGym.stationaryBike) //

console.log(boxingGym.treadmill) //

```

[Run](#)

[Reset](#)

In this example, there are three classes defined: `StationaryBike`, `Treadmill`, and `Gym`. The `StationaryBike` class is coded so that its future object instance will have the `position` and `gears` properties. The `position` property describes where the stationary bike will be placed inside the gym, and the `gears` property gives the number of gears that that stationary bike should have. The `Treadmill` class also has a position, and another property, named `modes` (as in "exercise modes").

The `Gym` class has three parameters in its constructor function: `openHrs`, `stationaryBikePos`, `treadmillPos`.

This code allows me to instantiate a new instance object of the `Gym` class, and then when I inspect it, I get the following information:

- the `openHrs` property is equal to "7-22" (that is, 7am to 10pm)
- the `stationaryBike` property is an object of the `StationaryBike` type, containing two properties: `position` and `gears`
- the `treadmill` property is an object of the `Treadmill` type, containing two properties: `position` and `modes`

[Completed](#)

Default Parameters

A useful a ES6 feature allows me to set a default parameter inside a function definition First, . What that means is, I'll use an ES6 feature which allows me to set a default parameter inside a function definition, which goes hand in hand with the defensive coding approach, while requiring almost no effort to implement.

For example, consider a function declaration without default parameters set:

```
1  
2  
3  
  
function noDefaultParams(number) {  
  
    console.log('Result:', number * number)  
  
}
```

Obviously, the `noDefaultParams` function should return whatever number it receives, *squared*. However, what if I call it like this:

```
1  
2  
3  
  
noDefaultParams(); // Result: NaN
```

JavaScript, due to its dynamic nature, doesn't throw an error, but it does return a non-sensical output.

Consider now, the following improvement, using default parameters:

```
1  
2  
3  
  
function withDefaultParams(number = 10) {
```

```
        console.log('Result:', number * number)

    }
```

Default params allow me to build a function that will run with default argument values even if I don't pass it any arguments, while still being flexible enough to allow me to pass custom argument values and deal with them accordingly.

This now allows me to code my classes in a way that will promote easier object instantiation.

Consider the following class definition:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

class NoDefaultParams {
```

```

constructor(num1, num2, num3, string1, bool1) {

    this.num1 = num1;

    this.num2 = num2;

    this.num3 = num3;

    this.string1 = string1;

    this.bool1 = bool1;

}

calculate() {

    if(this.bool1) {

        console.log(this.string1, this.num1 + this.num2 + this.num3);

        return;
    }

    return "The value of bool1 is incorrect"
}

}

```

Now I'll instantiate an object of the `NoDefaultParams` class, and run the `calculate()` method on it. Obviously, the `bool1` should be set to `true` on invocation to make this work, but I'll set it to false on purpose, to highlight the point I'm making.

1

2

```

var fail = new NoDefaultParams(1,2,3,false);

fail.calculate(); // 'The value of bool1 is incorrect'

```

This example might highlight the reason sometimes weird error messages appear when some software is used - perhaps the developers just didn't have enough time to build it better.
However, now that you know about default parameters, this example can be improved as follows:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
class WithDefaultParams {
```

```

    constructor(num1 = 1, num2 = 2, num3 = 3, string1 = "Result:", bool1 =
true) {

    this.num1 = num1;

    this.num2 = num2;

    this.num3 = num3;

    this.string1 = string1;

    this.bool1 = bool1;

}

calculate() {

    if(this.bool1) {

        console.log(this.string1, this.num1 + this.num2 + this.num3);

        return;
    }

    return "The value of bool1 is incorrect"
}

var better = new WithDefaultParams();

better.calculate(); // Result: 6

```

This approach improves the developer experience of my code, because I no longer have to worry about feeding the `WithDefaultParameters` class with all the arguments. For quick tests, this is great, because I no longer need to worry about passing the proper arguments.

Additionally, this approach really shines when building inheritance hierarchies using classes, as it makes it possible to provide only the custom properties in the sub-class, while still accepting all the default parameters from the super-class constructor.

In conclusion, in this reading I've covered the following:

- How to approach designing an object-oriented program in JavaScript
- The role of the `extends` and `super` keywords
- The importance of using default parameters.

Completed

Designing an OO Program

In this reading, I will show you how to create classes in JavaScript, using all the concepts you've learned so far.

Specifically, I'm preparing to build the following inheritance hierarchy:

Animal / \ Cat Bird / \\ HouseCat Tiger Parrot

There are two keywords that are essential for OOP with classes in JavaScript.

These keywords are `extends` and `super`.

The `extends` keyword allows me to inherit from an existing class.

Based on the above hierarchy, I can code the `Animal` class like this:

```
1  
2  
3  
  
class Animal {  
  
    // ... class code here ...  
  
}
```

Then I can code, for example, the `Cat` sub-class, like this:

```
1  
2  
3  
  
class Cat extends Animal {  
  
    // ... class code here ...
```

```
}
```

This is how the `extends` keyword is used to setup inheritance relationships.

The `super` keyword allows me to "borrow" functionality from a super-class, in a sub-class. The exact dynamics of how this works will be covered later on in this lesson.

Now I can start thinking about how to implement my OOP class hierarchy.

Before I even begin, I need to think about things like: * What should go into the base class of `Animal`? In other words, what will all the sub-classes inherit from the base class? * What are the specific properties and methods that separate each class from others? * Generally, how will my classes relate to one another?

Once I've thought it through, I can build my classes.

So, my plan is as follows:

1. The `Animal` class' constructor will have two properties: `color` and `energy`
2. The `Animal` class' prototype will have three methods: `isActive()`, `sleep()`, and `getColor()`.
3. The `isActive()` method, whenever ran, will lower the value of `energy` until it hits 0. The `isActive()` method will also report the updated value of `energy`. If `energy` is at zero, the animal object will immediately go to sleep, by invoking the `sleep()` method based on the said condition.
4. The `getColor()` method will just console log the value in the `color` property.
5. The `Cat` class will inherit from `Animal`, with the additional `sound`, `canJumpHigh`, and `canClimbTrees` properties specific to the `Cat` class. It will also have its own `makeSound()` method.
6. The `Bird` class will also inherit from `Animal`, but its own specific properties will be quite different from `Cat`. Namely, the `Bird` class will have the `sound` and the `canFly` properties, and the `makeSound` method too.
7. The `HouseCat` class will extend the `Cat` class, and it will have its own `houseCatSound` as its special property. Additionally, it will override the `makeSound()` method from the `Cat` class, but it will do so in an interesting way. If the `makeSound()` method, on invocation, receives a single `option` argument - set to `true`, then it will run `super.makeSound()` - in other words, run the code from the parent class (`Cat`) with the addition of running the `console.log(this.houseCatSound)`. Effectively, this means that the `makeSound()` method on the `HouseCat` class' instance object will have two separate behaviors, based on whether we pass it `true` or `false`.
8. The `Tiger` class will also inherit from `Cat`, and it will come with its own `tigerSound` property, while the rest of the behavior will be pretty much the same as in the `HouseCat` class.
9. Finally, the `Parrot` class will extend the `Bird` class, with its own `canTalk` property, and its own `makeSound()` method, working with two conditionals: one that checks if the value of `true` was passed to `makeSound` during invocation, and another that checks the value stored inside `this.canTalk` property.

Now that I have fully explained how all the code in my class hierarchy should work I might start implementing it by adding all the requirements as comments inside the code structure.

At this stage, with all the requirements written down as comments, my code should be as follows:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

```
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

class Animal {
    // constructor: color, energy
    // isActive()
    // if energy > 0, energy -=20, console log energy
    // else if energy <= 0, sleep()
```

```
// sleep()

// energy += 20

// console.log energy

}

class Cat extends Animal {

// constructor: sound, canJumpHigh, canClimbTrees, color, energy

// makeSound()

// console.log sound

}

class Bird extends Animal {

// constructor: sound, canFly, color, energy

// makeSound()

// console.log sound

}

class HouseCat extends Cat {

// constructor: houseCatSound, sound, canJumpHigh, canClimbTrees,
color, energy

// makeSound(option)

// if (option)

// super.makeSound()

// console.log(houseCatSound)
```

```

}

class Tiger extends Cat {

    // constructor: tigerSound, sound, canJumpHigh, canClimbTrees, color,
energy

    // makeSound(option)

    // if (option)

        // super.makeSound()

        // console.log(tigerSound)

}

class Parrot extends Bird {

    // constructor: canTalk, sound, canJumpHigh, canClimbTrees, color,
energy

    // makeSound(option)

    // if (option)

        // super.makeSound()

        // if (canTalk)

            // console.log("talking!")

```

Now that I've coded my requirements inside comments of otherwise empty classes, I can start coding each class as per my specifications.

Coding the Animal class

First, I'll code the base **Animal** class.

1

2

3

```
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

class Animal {
    constructor(color = 'yellow', energy = 100) {
        this.color = color;
    }
}
```

```

    this.energy = energy;

}

isActive() {

    if(this.energy > 0) {

        this.energy -= 20;

        console.log('Energy is decreasing, currently at:',
this.energy)

    } else if(this.energy == 0) {

        this.sleep();

    }

}

sleep() {

    this.energy += 20;

    console.log('Energy is increasing, currently at:', this.energy)

}

getColor() {

    console.log(this.color)

}

}

```

Each animal object, no matter which one it is, will share the properties of `color` and `energy`. Now I can code the `Cat` and `Bird` classes:

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

```
class Cat extends Animal {

    constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true,
color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canClimbTrees = canClimbTrees;

        this.canJumpHigh = canJumpHigh;

    }

    makeSound() {

        console.log(this.sound);

    }

}

class Bird extends Animal {

    constructor(sound = 'chirp', canFly = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canFly = canFly;

    }

    makeSound() {

        console.log(this.sound);

    }

}
```

```
    }  
  
}
```

Note: If I didn't use the `super` keyword in our sub-classes, once I'd run the above code, I'd get the following error: `Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor.`
And now I can code the three remaining classes: `HouseCat`, `Tiger`, and `Parrot`.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

```
class HouseCat extends Cat {  
  
    constructor(houseCatSound = "meow", sound, canJumpHigh, canClimbTrees,  
color, energy) {  
  
        super(sound, canJumpHigh, canClimbTrees, color, energy);  
  
        this.houseCatSound = houseCatSound;  
  
    }  
  
    makeSound(option) {  
  
        if (option) {  
  
            super.makeSound();  
  
        }  
  
        console.log(this.houseCatSound);  
  
    }  
  
}  
  
class Tiger extends Cat {  
  
    constructor(tigerSound = "Roar!", sound, canJumpHigh, canClimbTrees,  
color, energy) {  
  
        super(sound, canJumpHigh, canClimbTrees, color, energy);  
  
        this.tigerSound = tigerSound;  
  
    }  
}
```


}

Now that we've set up this entire inheritance structure, we can build various animal objects. For example, I can build two parrots: one that can talk, and the other that can't.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

```
var polly = new Parrot(true); // we're passing `true` to the constructor
so that polly can talk

var fiji = new Parrot(false); // we're passing `false` to the constructor
so that fiji can't talk

polly.makeSound(); // 'chirp', 'I'm a talking parrot!'

fiji.makeSound(); // 'chirp'

polly.color; // yellow

polly.energy; // 100

polly.isActive(); // Energy is decreasing, currently at: 80

var penguin = new Bird("shriek", false, "black and white", 200); //
setting all the custom properties

penguin; // Bird {color: 'black and white', energy: 200, sound: 'shriek',
canFly: false }

penguin.sound; // 'shriek'

penguin.canFly; // false

penguin.color; // 'black and white'

penguin.energy; // 200

penguin.isActive(); // Energy is decreasing, currently at: 180
```

Also, I can build a pet cat:

1

```
var leo = new HouseCat();
```

Now I can have `leo` purr:

1

2

3

4

5

```
// leo, no purring please:
```

```
leo.makeSound(false); // meow
```

```
// leo, both purr and meow now:
```

```
leo.makeSound(true); // purr, meow
```

Additionally, I can build a tiger:

1

```
var cuddles = new Tiger();
```

My `cuddles` tiger can purr and roar, or just roar:

1

2

```
cuddles.makeSound(false); // Roar!
```

```
cuddles.makeSound(true); // purr, Roar!
```

Here's the complete code from this lesson, for easier copy-pasting:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

```
class Animal {  
  
    constructor(color = 'yellow', energy = 100) {  
  
        this.color = color;  
    }  
}
```

```
        this.energy = energy;

    }

isActive() {

    if(this.energy > 0) {

        this.energy -= 20;

        console.log('Energy is decreasing, currently at:', this.energy)

    } else if(this.energy == 0) {

        this.sleep();

    }

}

sleep() {

    this.energy += 20;

    console.log('Energy is increasing, currently at:', this.energy)

}

getColor() {

    console.log(this.color)

}

}

class Cat extends Animal {
```

```
    constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true,
color, energy) {

    super(color, energy);

    this.sound = sound;

    this.canClimbTrees = canClimbTrees;

    this.canJumpHigh = canJumpHigh;

}

makeSound() {

    console.log(this.sound);

}

}

class Bird extends Animal {

    constructor(sound = 'chirp', canFly = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canFly = canFly;

    }

}
```

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[Constructor](#)

[Classes](#)

[Object-oriented programming](#)

[Regular Expressions in JavaScript](#)

[RegExp object in JavaScript](#)

Completed

For of loops and objects

In this reading, you'll learn how the for of loop works conceptually.

To begin, it's important to know that a for of loop cannot work on an object directly, since **an object is not iterable**. For example:

1

2

3

4

5

6

7

8

```
const car = {
```

```
    speed: 100,
```

```
    color: "blue"
```

```
}
```

```
for(prop of car) {  
  
    console.log(prop)  
  
}
```

[Run](#)

[Reset](#)

Running the above code snippet will throw the following error:

[1](#)

`Uncaught TypeError: car is not iterable`

Contrary to objects, arrays *are* iterable. For example:

[1](#)

[2](#)

[3](#)

[4](#)

```
const colors = ['red', 'orange', 'yellow']  
  
for (var color of colors) {  
  
    console.log(color);  
  
}
```

[Run](#)

[Reset](#)

This time, the output is as follows:

[1](#)

[2](#)

red

orange

yellow

Luckily, you can use the fact that a for of loop can be run on arrays *to loop over objects*.

But how?

Before you can properly answer this question, you first need to review three built-in methods:

`Object.keys()`, `Object.values()`, and `Object.entries()`.

Built-in methods

The `Object.keys()` method

The `Object.keys()` method receives an object as its parameter. Remember, this object is **the object you want to loop over**. It's still too early to explain how you'll loop over the object itself; for now, focus on the returned array of properties when you call the `object.keys()` method.

Here's an example of running the `Object.keys()` method on a brand new `car2` object:

```
1
2
3
4
5

const car2 = {
  speed: 200,
  color: "red"
}

console.log(Object.keys(car2)); // ['speed', 'color']
```

Run

Reset

So, when I run `object.keys()` and pass it my `car2` object, **the returned value is an array of strings**, where each string is a property key of the properties contained in my `car2` object.

The `Object.values()` method

Another useful method is `Object.values()`:

```
1  
2  
3  
4  
5  
  
const car3 = {  
  
    speed: 300,  
  
    color: "yellow"  
  
}  
  
console.log(Object.values(car3)); // [300, 'yellow']
```

The `Object.entries()` method

Finally, there's another useful method, `Object.entries()`, which returns an array listing both the keys and the values.

```
1  
2  
3  
4  
5  
  
const car4 = {  
  
    speed: 400,
```

```
color: 'magenta'

}

console.log(Object.entries(car4));
```

What gets returned from the invocation of the `object.entries()` method is the following:

[1](#)

```
[ ['speed', 400], ['color', 'magenta'] ]
```

This time, the values that get returned are 2-member arrays nested inside an array. In other words, you get an array of arrays, where each array item has two members, the first being a property's key, and the second being a property's value.

Effectively, it's as if you was listing all of a given object's properties, a bit like this:

[1](#)

[2](#)

[3](#)

[4](#)

[5](#)

```
[

[propertyKey, propertyVal],
[propertyKey, propertyVal],
...etc

]
```

To summarise, you learned that you can loop over arrays using the `for of` loop. You also learned that you can extract object's keys, values, or both, using the `Object.keys()`, `Object.values()` and `Object.entries()` syntax.

Examples

You now have all the ingredients that you need to **loop over any object's own property keys and values**.

Here's a very simple example of doing just that:

```
1
2
3
4
5
6
7
8
9
10

var clothingItem = {

  price: 50,

  color: 'beige',

  material: 'cotton',

  season: 'autumn'

}

for( key of Object.keys(clothingItem) ) {

  console.log(keys, ":", clothingItem[key])

}
```

The trickiest part to understand in this syntax is probably the `clothingItem[key]`.

Luckily, this is not too hard to comprehend, especially since you've already covered the concept previously when you were learning **how to access an object's member using the brackets notation**.

Recall that you also learned how you can dynamically access a property name.

To revisit this concept and show a practical demo of how that works, let's code a function declaration that randomly assigns either the string `speed` or the string `color` to a variable name, and then build an object that has only two keys: a `speed` key and a `color` key.

After this setup, you will be able to dynamically access either one of those properties on a brand new `drone` object, using the brackets notation.

Here's the example's code:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16
```

```
function testBracketsDynamicAccess() {  
  
  var dynamicKey;  
  
  if (Math.random() > 0.5) {  
  
    dynamicKey = "speed";  
  
  } else {  
  
    dynamicKey = "color";  
  
  }  
  
  var drone = {  
  
    speed: 15,  
  
    color: "orange"  
  
  }  
  
  console.log(drone[dynamicKey]);  
  
}  
  
testBracketsDynamicAccess();
```

[Run](#)

[Reset](#)

This example might feel a bit convoluted, but its purpose is to demo the fact that you're getting either one or the other value from an object's key, based on the string that got assigned to the `dynamicKey` variable, and accessed without issues, using the brackets notation.

Feel free to run the `testBracketsDynamicAccess()` function a few times, and you'll notice that sometimes the value that gets output is `15`, and sometimes it's `orange`, although I'm always accessing the `drone[dynamicKey]` key. Since the value of the `dynamicKey` is populated on the

`Math.random()` invocation, sometimes that expression evaluates to `drone["speed"]`, and other times that expression evaluates to `drone["color"]`.

You have now learned about the building blocks that make the for of loop useful to iterate over objects - *although objects are not iterables*.

Next, you'll have a go at a practical example of working with both the for of and the for in loop. Each loops have their place and can be considered useful in different situations.

Completed

Template literals examples

The aim of this reading is to help you understand how template literals work.

What are template literals?

Template literals are an alternative way of working with strings, which was introduced in the ES6 addition to the JavaScript language.

Up until ES6, the only way to build strings in JavaScript was to delimit them in either single quotes or double quotes:

1

2

```
'Hello, World!'
```

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

Alongside the previous ways to build strings, ES6 introduced the use of backtick characters as delimiters:

1

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

The above code snippet is an example of a template string, which is also known as a template literal.
Note: On most keyboards, the backtick character can be located above the TAB key, to the left of the number 1 key.

Differences between a template and regular string

There are several ways in which a template string is different from a regular string.

- First, it allows for **variable interpolation**:

1

2

3

```
let greet = "Hello";  
  
let place = "World";  
  
console.log(` ${greet} ${place} ! `) //display both variables using template literals
```

The above console log will output:

1

Hello World !

Essentially, using template literals allows programmers to embed variables directly in between the backticks, without the need to use the + operator and the single or double quotes to delimit string literals from variables. In other words, in ES5, the above example would have to be written as follows:

1

2

3

```
var greet = "Hello";  
  
var place = "World";  
  
console.log(greet + " " + place + " ! ") //display both variables without using template literals
```

- Besides variable interpolation, template strings can span multiple lines.

For example, this is perfectly good syntax:

1

2

3

```
`Hello,
```

```
World
```

```
!
```

```
'
```

Notice that this can't be done using **string literals** (that is, strings delimited in single or double quotes):

1

2

```
"Hello,
```

```
World"
```

The above code, when run, will throw a syntax error.

Put simply, template literals allow for multi-line strings - something that simply isn't possible with string literals.

- Additionally, the reason why it's possible to interpolate variables in template literals is because this syntax actually allows for **expression evaluation**.

In other words, this:

1

2

```
//it's possible to perform arithmetic operation inside a template literal
expression
```

```
console.log(`$1 + 1 + 1 + 1 + 1} stars!`)
```

The above example will console log the following string: `5 stars!`.

This opens up a host of possibilities. For example, it's possible to evaluate a ternary expression inside a template literal.

Some additional use cases of template literals are **nested template literals** and **tagged templates**. However, they are a bit more involved and are beyond the scope of this reading.

If you're curious about how they work, please refer to the additional reading provided at the end of this lesson.

Completed

Data Structures examples

In this reading, you'll learn about some of the most common examples of data structures.

The focus will be on working with the Object, Array, Map and Set data structures in JavaScript, through a series of examples.

Note that this reading will not include a discussion of some data structures that exist in some other languages, such as Queues or Linked Lists. Although these data structures (and other data structures too!) can be custom-coded in JavaScript, the advanced nature of these topics and the difficulty with implementing related exercises means they are beyond the scope of this reading.

Working with arrays in JavaScript

Previously, you've covered a lot of concepts related to how to work with JavaScript arrays.

However, there are still a few important topics that can be covered, and one of those is, for example, working with some built-in methods.

In this reading, the focus is on three specific methods that exist on arrays:

1. `forEach`
2. `filter`
3. `map`

Let's explore these methods.

The `forEach()` method

Arrays in JavaScript come with a handy method that allows you to loop over each of their members.

Here's the basic syntax:

1

2

3

4

5

```
const fruits = ['kiwi', 'mango', 'apple', 'pear'];

function appendIndex(fruit, index) {

    console.log(` ${index}. ${fruit}`)
```

```
}

fruits.forEach(appendIndex);
```

The result of running the above code is this:

- 1
- 2
- 3
- 4
- 0. kiwi
- 1. mango
- 2. apple
- 3. pear

To explain the syntax, the `forEach()` method accepts a **function that will work on each array item**. That function's first parameter is the current array item itself, and the second (optional) parameter is the index.

Very often, the function that the `forEach()` method needs to use is passed in directly into the method call, like this:

```
const veggies = ['onion', 'garlic', 'potato'];

veggies.forEach( function(veggie, index) {

    console.log(` ${index}. ${fruit}`);
}) ;
```

This makes for more compact code, but perhaps somewhat harder to read. To increase readability, sometimes arrow functions are used. You can find out more about arrow functions in the additional reading.

The `filter()` method

Another very useful method on the array is the `filter()` method. It filters your arrays **based on a specific test**. Those array items that pass the test are returned.

Here's an example:

```
1  
2  
3  
4  
  
const nums = [0,10,20,30,40,50];  
  
nums.filter( function(num) {  
  
    return num > 20;  
  
})
```

Here's the returned array value:

```
[30,40,50]
```

Similar to the `forEach()` method, the `filter()` method also accepts a function and that function performs some work on each of the items in the array.

The `map` method

Finally, there's a very useful `map` method.

This method is used to map each array item over to another array's item, based on whatever work is performed inside the function that is passed-in to the `map` as a parameter.

For example:

```
1  
2  
3
```

```
[0,10,20,30,40,50].map( function(num) {  
  return num / 10  
})
```

The return value from the above code is:

[1, 2, 3, 4, 5]

As already discussed, choosing a proper data structure affects the very code that you can write. This is because the data structure itself comes with some built-in functionality that makes it easier to perform certain tasks or makes it harder or even impossible without converting your code to a proper data structure.

Now that you've covered the methods, let's explore how to work with different built-in data structures in JavaScript.

Working with Objects in JavaScript

A lot of the information on how to work with objects in JavaScript has already been covered in this course.

The example below demonstrates how to use the object data structure to complete a specific task. This task is to convert an object to an array:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

```
const result = [];

const drone = {
    speed: 100,
    color: 'yellow'
}

const droneKeys = Object.keys(drone);

droneKeys.forEach( function(key) {
    result.push(key, drone[key])
})

console.log(result)
```

This is the result of executing the above code:

[1](#)

```
['speed', 100, 'color', 'yellow']
```

Although this is possible and works, having to do something like this might mean that you haven't chosen the correct data structure to work with in your code.

On the flip side, sometimes you don't get to pick the data structure you're working with. Perhaps that data comes in from a third-party data provider and all you can do is code your program so that it consumes it. You'll learn more about the interchange of data on the web when you learn about JSON (JavaScript Object Notation).

Working with Maps in JavaScript

To make a new Map, you can use the `Map` constructor:

[1](#)

```
new Map();
```

A map can feel very similar to an object in JS.

However, it doesn't have inheritance. No prototypes! This makes it useful as a data storage.

For example:

[1](#)

2

3

4

5

6

```
let bestBoxers = new Map();

bestBoxers.set(1, "The Champion");

bestBoxers.set(2, "The Runner-up");

bestBoxers.set(3, "The third place");

console.log(bestBoxers);
```

Here's the console output:

1

```
Map(3) {1 => 'The Champion', 2 => 'The Runner-up', 3 => 'The third place'}
```

To get a specific value, you need to use the `get()` method. For example:

1

```
bestBoxers.get(1); // 'The Champion'
```

Working with Sets in JavaScript

A set is a collection of unique values.

To build a new set, you can use the `Set` constructor:

1

```
new Set();
```

The `Set` constructor can, for example, accept an array.

This means that we can use it to quickly filter an array for unique members.

1

2

3

```
const repetitiveFruits = ['apple', 'pear', 'apple', 'pear', 'plum', 'apple'];

const uniqueFruits = new Set(repetitiveFruits);

console.log(uniqueFruits);
```

The above code outputs the following in the console:

1

```
{'apple', 'pear', 'plum'}
```

Other data structures in JavaScript

Besides the built-in data structures in JavaScript, it's possible to build non-native, custom data structures.

These data structures come built-in natively in some other programming languages or even those other programming languages don't support them natively.

Some more advanced data structures that have not been covered include:

- Queues
- Linked lists (singly-linked and doubly-linked)
- Trees
- Graphs

For resources on building these data structures, please refer to the additional reading.

Completed

Using Spread and Rest

In this reading, you'll learn how to join arrays, objects using the rest operator. You will also discover how to use the spread operator to:

- Add new members to arrays without using the `push()` method,
- Convert a string to an array and
- Copy either an object or an array into a separate object

Recall that the `push()` and `pop()` methods are used to add and remove items from the end of an array.

Join arrays, objects using the rest operator

Using the spread operator, it's easy to concatenate arrays:

```
1  
2  
3  
4  
  
const fruits = ['apple', 'pear', 'plum']  
  
const berries = ['blueberry', 'strawberry']  
  
const fruitsAndBerries = [...fruits, ...berries] // concatenate  
  
console.log(fruitsAndBerries); // outputs a single array
```

Here's the result:

```
1  
  
['apple', 'pear', 'plum', 'blueberry', 'strawberry']
```

It's also easy to join objects:

```
1  
2  
3  
4  
  
const flying = { wings: 2 }  
  
const car = { wheels: 4 }  
  
const flyingCar = {...flying, ...car}  
  
console.log(flyingCar) // {wings: 2, wheels: 4}
```

Add new members to arrays without using the `push()` method

Here's how to use the spread operator to easily add one or more members to an existing array:

```
1  
2  
3  
4  
  
let veggies = ['onion', 'parsley'];  
  
veggies = [...veggies, 'carrot', 'beetroot'];  
  
console.log(veggies);
```

Here's the output:

```
1  
2  
3  
4  
['onion', 'parsley', 'carrot', 'beetroot']
```

Convert a string to an array using the spread operator

Given a string, it's easy to spread it out into separate array items:

```
1  
2  
3  
  
const greeting = "Hello";  
  
const arrayOfChars = [...greeting];  
  
console.log(arrayOfChars); // ['H', 'e', 'l', 'l', 'o']
```

Copy either an object or an array into a separate one

Here's how to copy an object into a completely separate object, using the spread operator.

```
1
2
3
4
5
6
7
8
9

const car1 = {
  speed: 200,
  color: 'yellow'
}

const car2 = {...car1}

car1.speed = 201

console.log(car1.speed, car2.speed)
```

The output is 201, 200.

You can copy an array into a completely separate array, also using the spread operator, like this:

```
1
2
```

3

4

```
const fruits1 = ['apples', 'pears']

const fruits2 = [...fruits]

fruits1.pop()

console.log(fruits1, "not", fruits2)
```

This time, the output is:

1

```
['apples'] 'not' ['apples', 'pears']
```

Note that the spread operator only performs a shallow copy of the source array or object. For more information on this, please refer to the additional reading.

There are many more tricks that you can perform with the spread operator. Some of them are really handy when you start working with a library such as React.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[Template literals](#)

[Arrow functions](#)

[Spread syntax](#)

[Rest parameters](#)

[JavaScript data structures](#)

Completed

JavaScript interactivity

The purpose of this reading is to introduce you to a simple explanation of web page manipulation and some examples of it.

Did you know that JavaScript's initial purpose was to **provide interactivity in the browser?**

In other words, it was the "set of controls" that would allow web developers to control the behavior of the webpages and even the browsers that these webpages worked on.

This is still the case today.

As the web ecosystem developed and the world became ever more digitized, so did the myriad of ways that one can use JavaScript as a web developer to manipulate websites.

Initially, in the late 1990s, there was plain JavaScript that had to be tweaked to suit individual browsers.

Then, by the mid-2000s, the jQuery library came out, with the idea of writing less code, but doing more with it. It "leveled the playing field" as it allowed developers to use a single code-base for various browsers.

This trend continued and many other frameworks such as React, Vue, Angular, D3, and more came along.

Together with npm and Node.js, the JavaScript ecosystem is not slowing down.

Even though it has gone a long way, sometimes it's good to go back to basics. JavaScript is still the king when it comes to making our websites interactive.

Although CSS has developed significantly over the years, it is still JavaScript that allows users to:

- Get their geolocation,
- Interact with maps,
- Play games in the browser,
- Handle all kinds of user-triggered events, regardless of the device,
- Verify form input before sending it to the backend of a webapp,
- and more!

There are many, many ways in which JavaScript allows you to build rich, interactive experiences on the web.

Completed

Exercise: Web page content update

In this reading, you will learn how to capture user input and process it. You'll be introduced to a simple example that demonstrates how to manipulate information displayed based on user input. To capture user input, you can use the built-in `prompt()` method, like this:

1

```
let answer = prompt('What is your name?');
```

Once you have the user-provided input inside the `answer` variable, you can manipulate it any way you need to.

For example, you can output the typed-in information on the screen, as an `<h1>` HTML element.

Here's how you'd do that:

```
1  
2  
3  
4  
5  
6  
7  
  
let answer = prompt('What is your name?');  
  
if (typeof(answer) === 'string') {  
  
  var h1 = document.createElement('h1')  
  
  h1.innerText = answer;  
  
  document.body.innerText = '';  
  
  document.body.appendChild(h1);  
  
}  
  
}
```

This is probably the quickest and easiest way to capture user input on a website, but doing it this way is not the most efficient approach, especially in more complex scenarios.

This is where HTML forms come in.

You can code a script which will take an input from an HTML form and display the text that a user types in on the screen.

Here's how this is done.

You'll begin by coding out a "test solution" to the task at hand:

1

2

3

4

5

6

7

8

9

```
var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"
```

```
var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);
```

So, you're essentially doing the same thing that you did before, only this time you're also dynamically adding the `input` element, and you're setting its HTML `type` attribute to `text`. That way, when you start typing into it, the letters will be showing in the `h1` element above.

However, you're not there quite yet. At this point, the code above, when run on a live website, will add the `h1` element with the text "Type into the input to make this text change", and an empty input form field under it.

You can try this code out yourself, for example, by pointing your browser to the `example.com` website, and running the above code in the console.

Remember you can access the console from the developer tools in your browser.

Another opinionated thing that you did in the code above is: setting my variables using the `var` keyword.

Although it's better to use either `let` or `const`, you're just running a quick experiment on a live website, and you want to use the most lenient variable keyword, the one which will not complain about you having already set the `h1` or the `input` variables.

If you had a complete project with a modern JavaScript tooling setup, you'd be using `let` or `const`, but this is just a quick demo, so using `var` in this case is ok.

The next thing that you need to do is: set up an event listener. The event you're listening for is the `change` event. In this case, the change event will fire after you've typed into the input and pressed the ENTER key.

Here's your updated code:

```
1
2
3
4
5
6
7
8
9
10
11
12
13

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"
```

```
var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

input.addEventListener('change', function() {

    console.log(input.value)

})
```

This time, when you run the above code on the said `example.com` website, subsequently typing some text into the input field and pressing the enter key, you'll get the value of the typed-in text logged to the console.

Now, the only thing that you still need to do to complete my code is to update the text content of the `h1` element with the value you got from the `input` field.

Here's the complete, updated code:

1
2
3
4
5
6
7
8

9

10

11

12

13

```
var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

input.addEventListener('change', function() {

    h1.innerText = input.value

})
```

After this update, whatever you type into the input, after pressing the ENTER key, will be shown as the text inside the `h1` element.

Although this completes this lesson item, it's important to note that combining DOM manipulation and event handling allows for some truly remarkable interactive websites.

Completed

Exercise: Capture Data

Description

The aim of this exercise is to access the content of an element, specifically to use a button click to replace text.

Task 1: The example.com website

Open the [example.com](http://www.example.com) website in your browser. Open the developer tools and focus on the Console tab.

Example.com is a domain that can be used as an example in documents, papers and websites. If you navigate in your browser to <http://www.example.com> you will see a webpage with a simple message:

Example Domain

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.

Task 2: Get h1 into a variable

Use the `document.querySelector()` method to query the h1 element on the page and assign it to the variable named `h1`.

Task 3: Code an array

Declare a new variable, name it `arr`, and save the following array into it:

1

2

3

4

5

6

```
[  
    'Example Domain',  
    'First Click',  
    'Second Click',  
    'Third Click'  
]
```

Run

Reset

Task 4: Write a click-handling function

Write a new function declaration, named `handleClicks`. It should not accept any parameters.

Inside of it, code a `switch` statement, and pass a single parameter to it, `h1.innerText`.

The body of the switch statement should have a total of 4 cases (the fourth being the default case).

The first case should start with `case arr[0]:`. It should set the `h1.innerText` to `arr[1]`. In other words, it should assign the value of `arr[1]` to the `h1.innerText` property. The next line should have only the `break` keyword.

The second case should start with `case arr[1]:`. It should set the `h1.innerText` to `arr[2]`. In other words, it should assign the value of `arr[2]` to the `h1.innerText` property. The next line should have only the `break` keyword.

The third case should start with `case arr[2]:`. It should set the `h1.innerText` to `arr[3]`. In other words, it should assign the value of `arr[3]` to the `h1.innerText` property. The next line should have only the `break` keyword.

The `default` case should set the value of the `h1.innerText` property to `arr[0]`.

Task 5: Add an event listener

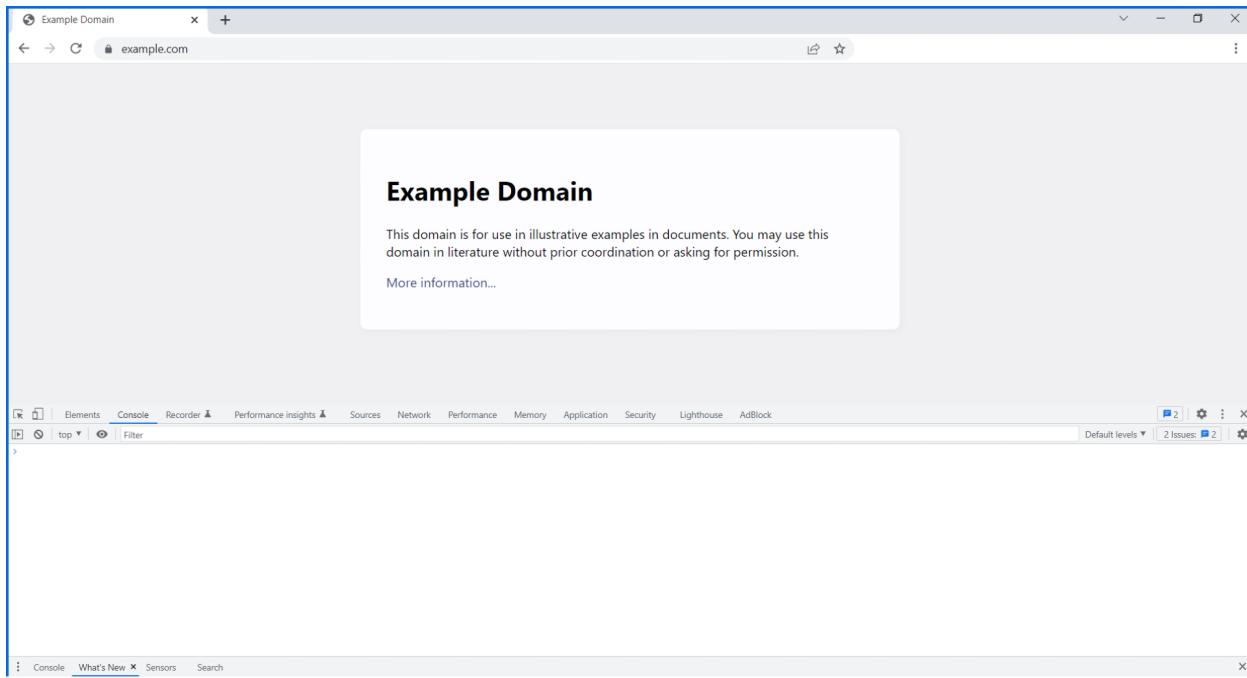
You've created an `h1` variable in Task 2. Now, use that variable to run the `addEventListener()` method on it. Pass two arguments to the `addEventListener()` method: '`click`' and `handleClicks`.

Completed

Solution: Capture Data

Task 1 solution: The example.com website

1. Open your favorite browser and navigate to <https://example.com/>.
2. Next open its developer tools using either the F12 key, or right-clicking onto the page and selecting the contextual menu's Inspect command, or by pressing CTRL SHIFT I or COMMAND SHIFT I.
3. Next, click on the Console tab to open it in a dedicated tab, or press the ESC key to have the console open while any tab is in focus.



Task 2 solution: Get h1 into a variable

1

2

```
var h1 = document.querySelector('h1')
```

[Run](#)

[Reset](#)

Task 3 solution: Code an array

```
1  
2  
3  
4  
5  
6  
7  
  
var arr = [  
    'Example Domain',  
    'First Click',  
    'Second Click',  
    'Third Click'  
]
```

[Run](#)

[Reset](#)

Task 4 solution: Write a click-handling function

1

2

```
3
4
5
6
7
8
9
10
11
12
13
14
15

function handleClicks() {
    switch(h1.innerText) {
        case arr[0]:
            h1.innerText = arr[1]
            break
        case arr[1]:
            h1.innerText = arr[2]
            break
    }
}
```

```
    case arr[2]:  
  
        h1.innerText = arr[3]  
  
        break  
  
    default:  
  
        h1.innerText = arr[0]  
  
    }  
  
}  
  
Run
```

Reset

Task 5 solution: Add an event listener

1

2

```
h1.addEventListener('click', handleClicks);
```

Run

Reset

An example of the solution being run in the browser:

Play Video

Completed

Moving data around on the web

The modern web consists of millions and millions of web pages, connected services and databases. There are websites communicating with other websites, getting data from data feeds and data providers, both paid and free.

All of these data flows must be facilitated with some kind of data format.

Around 2001, Douglas Crockford came up with a data interchange format based on JavaScript objects. The name given to this format was JSON, which is JavaScript Object Notation.

Before JSON, the most common data interchange file format was **XML** (Extensible Markup Language). However, due to XML's syntax, it required more characters to describe the data that was sent. Also, since it was a specific stand-alone language, it wasn't as easily inter-operable with JavaScript.

Thus, the two major reasons for the JSON format becoming the dominant data interchange format that it is today is two-fold:

- First, it's very lightweight, with syntax very similar to "a stringified JavaScript object". You'll learn more about the specifics of this later.
- Second, it's easier to handle in JavaScript code, since, JSON, after all, is *just JavaScript*.

It is often said that JSON is a *subset* of JavaScript, meaning it adheres to syntax rules of the JavaScript language, but it's even more strict in how proper JSON code should be formatted. In other words, all JSON code is JavaScript, but not all JavaScript code is JSON.

Besides being a data interchange format, JSON is also a file format. It's not uncommon to access some third-party data from a third-party website into our own code in the form of a `json` file.

For example, if you had a website with the data on stock price movements, you might want to get the data of the current stock prices from a data vendor. They might offer their data service by giving you access to the file named, say `stockPrices.json`, that you could access from their servers.

Once you'd downloaded that stringified JSON data into your own code, you could then convert that data to a plain JavaScript object.

That would mean that you could use your web application's code to "dig into" the third-party data-converted-to-a-JavaScript-object, so as to obtain specific information based on a given set of criteria.

For example, if the stringified JSON data was converted to an object that had the following structure:

1

2

3

4

5

6

7

```
8
9
10
11
12
13

const currencyInfo = {

  [
    USD: {
      // ...
    },
    GBP: {
      // ...
    },
    EUR: {
      // ...
    }
  ]
}
```

You could then access only the data on the `USD` property, if that's what was needed by your app at a given point in time.

Hopefully, with this explanation, you understand, at a high level, how and why you might want to use JSON in your own code.

It's all about getting stringified JSON data from a server, converting ("parsing") that data into JS objects in your own code, working with the converted data in your own code, and perhaps even stringifying the result into JSON, so that this data is then ready to, for example, be sent back to a server after your code has processed it locally.

JSON is just a string - but there are rules that it must follow

JSON is a string, but it must be a properly-formatted string. In other words, it must adhere to some rules.

If a JSON string is not properly formatted, JavaScript would not be able to parse it into a JavaScript object.

JSON can work with some primitives and some complex data types, as described below.

Only a subset of values in JavaScript can be properly stringified to JSON and parsed from a JavaScript object into a JSON string.

These values include:

- primitive values: strings, numbers, booleans, null
- complex values: objects and arrays (no functions!)
- Objects have double-quoted strings for all keys
- Properties are comma-delimited both in JSON objects and in JSON arrays, just like in regular JavaScript code
- String properties must be surrounded in double quotes. For example:

```
"fruits",
```

```
"vegetables"
```

- Number properties are represented using the regular JavaScript number syntax; e.g

```
5,
```

```
10,
```

```
1.2
```

- Boolean properties are represented using the regular JavaScript boolean syntax, that is:

```
true
```

and

```
false
```

- Null as a property is the same as in regular JavaScript; it's just a

```
null
```

You can use object literals and array literals, as long as you follow the above rules

What happens if you try to stringify a data type which is not accepted in JSON syntax?

For example, what if you try to stringify a function? **The operation will silently fail.**

If you try to stringify some other data types, such as a BigInt number, say `123n`, you'd get the following error: `Uncaught TypeError: Do not know how to serialize a BigInt.`

Some examples of JSON strings

Finally, here is an example of a stringified JSON object, with a single key-value pair:

```
'{"color": "red"}'
```

Here's a bit more complex JSON object:

```
'{"color": "red", "nestedObject": { "color": "blue" } }'
```

The above JSON object encodes two properties:

- "color": "red"
- "nestedObject": { "color": "blue" }

It's also possible to have a JSON string encoding just an array:

```
'["one", "two", "three"]'
```

The above JSON string encodes an array holding three items, three values of the string data type.

Obviously, just like objects, arrays can nest other simple or complex data structures.

For example:

```
'[{"color": "blue"}, {"color": "red"}]'
```

In the above example, the JSON string encodes an array which holds two objects, where each object consists of a single key-value pair, where both values are strings.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN: Modules](#)

[Nodejs.org official docs on CommonJS](#)

[MDN: DOM](#)

[MDN: Document.querySelector](#)

[MDN: Event](#)

[MDN: EventTarget.addEventListener](#)

[MDN: Working with JSON](#)

Installing Node and NPM

Before installing Node.js and npm on your machine, you first need to verify if it's already installed.

Verifying the existing installation on Windows

On Windows, you can use the **WINKEY+r** shortcut key, which opens the Run window. Inside the **open:** input of the Run window, type **cmd** and press the enter key. This will open the command prompt.

Inside the command prompt, type:

- `node --version`

If there is Node.js installed on your Windows OS, it will return a value similar to this:

- `v16.14.2`

Then you can confirm that you have npm as well, running this:

- `npm --version`

If npm is installed, you'll get output similar to this:

- `8.5.0`

Verifying the existing Node.js and npm installation on Ubuntu (Linux)

You can quickly open a new bash (terminal) window on Ubuntu by pressing the **CTRL+ALT+t** shortcut key.

In the bash window that opens, type:

- `node --version && npm --version`

Both version numbers should appear in the bash window.

Installing Node.js and npm

On Windows OS

In case Node.js and npm are not installed on your Windows OS, navigate to <https://nodejs.org>. Locate the big download button, listing the LTS version. As of May 2022, the LTS version available for download is 16.15.0.

On Mac OS - XCode

To install brew, you need to install Xcode first. Homebrew does not come with its own compiler and it needs Xcode installed for it to work correctly. To install Xcode do the following:

1. Open a terminal.
2. Run the following:
`shell xcode-select --install`
3. A popup will appear asking you to confirm the installation. Click on the Install button.
4. Agree to the license agreement.

brew

Macs do not come with package managers like most Linux distributions. To make up for this an external tool called brew was created. To install brew, go to the official website (<https://brew.sh/>) and copy the command provided, open a terminal and run the command :

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once Brew is installed you can run the following command in the terminal

```
brew install node
```

Homebrew will download and install the dependencies, once this is complete, confirm the installation by typing

```
node -v
```

This will display the Node.js version

Type :

```
npm -v
```

to display the NPM version number.

On Ubuntu

Use the **CTRL+ALT+t** shortcut key to open a new bash window, then run the following commands:

- `sudo apt update`
- `sudo apt install nodejs`

That's it, you should be all set.

For a more advanced set up and troubleshooting, please refer to the additional reading.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN: Server-side website programming](#)

[Nodejs.org docs website](#)

[Jest testing framework website](#)

[Cypress testing framework](#)

[npm website](#)

[Unit testing in JavaScript](#)

About the Little Lemon receipt maker exercise

In this exercise, you will work with some data provided as an array of objects, listing information about dishes available in the Little Lemon restaurant.

You will need to write a function declaration which will be able to do two things:

1. If the function is called with the argument `true`, it will output the names of the dishes and calculate their final price (including 20% tax)
2. If the function is called with the argument `false`, it will output the names of the dishes and give their prices without the additional tax

The expected outcome is that all the dishes' names and prices will be shown in the console output.

The text below shows the output that your code should produce:

```
Prices with 20% tax: Dish: Italian pasta Price (incl.tax): $ 11.46 Dish: Rice with veggies Price (incl.tax): $ 10.38 Dish: Chicken with potatoes Price (incl.tax): $ 18.66 Dish: Vegetarian Pizza Price (incl.tax): $ 7.74 Prices without tax: Dish: Italian pasta Price (incl.tax): $ 9.55 Dish: Rice with veggies Price (incl.tax): $ 8.65 Dish: Chicken with potatoes Price (incl.tax): $ 15.55 Dish: Vegetarian Pizza Price (incl.tax): $ 6.45
```

Completed

Next steps

Congratulations! You've taken another step toward improving your knowledge, skills, and qualifications. The next course is React Basics and in it, you'll develop a working knowledge of React. React is a powerful JavaScript library that you can use to build user interfaces for web and mobile applications (apps). In this course, you will explore the fundamental concepts that underpin the React library and learn the basic skills required to build a simple, fast, and scalable app. Be sure to take the next course. It's your chance to gain further insight into the world of software development.

Completed

Course4-HTML and CSS in depth

Course syllabus

Prerequisites

To take this course, you don't need any developer experience, but you must be eager to get started with coding.

Module 1

In the first module, you will start with an introduction to the course and gain insight into what your career path as an HTML and CSS developer might look like. You will also receive tips on how to take this course successfully. Then, you'll move on to semantic tags and the importance of taking a structured approach to creating a well-formed web page. After which you will learn about metadata and tags and how you can use them to influence your web page's ranking through Search Engine Optimization (SEO). Following this section, you will learn about user input and forms and you will create and test a form by yourself. In the final part of the module, you will learn about media elements and you'll learn how to embed video and audio on a web page.

After completing this module you should be able to:

- Use common semantic and meta tags to improve the accessibility, readability and SEO of a web page.
- Create commonly-used web page layouts and components.
- Create and test a form with client-side validation.
- Recognize server-side connections and the languages used to carry out requests and responses.
- Post form data to a server.
- Create a video and audio player that can rate the media played.

Module 2

In module 2, you will focus on CSS layouts, grids and flexboxes. You will learn about fundamental layouts that you can use to design a good web page. Next, you will learn about CSS selectors which correspond to specific elements or element groups in an HTML document. In this section, you also learn about pseudo-class selectors that you can use to improve the interactivity of your web pages without having to add overly advanced styling. In this module, you will also learn about keyframes, animations and effects in CSS. The final part of this module is about how to use browser developer tools to assist with debugging and resolving HTML and CSS issues.

After completing this module you should be able to:

- Use Flexbox and CSS grids to create responsive layouts and charts.
- Use advanced CSS selectors such as pseudo-classes for targeted styling.
- Use CSS effects to introduce text effects, animations and transformations into your stylesheet.
- Create simple keyframe animations.
- Perform basic front-end testing, debugging and error handling.
- Adapt your CSS to perform as expected in different browsers.

Module 3

In the last module, you will have an opportunity to recap what you learned in the course and put it into practice by creating a home page for a client persona.

After completing this module you should be able to:

- Apply the skills you learned in this course to introduce more advanced styling into your portfolio.

Completed

How to be successful in this course

Taking an online course can be overwhelming. How do you learn at your own pace and successfully achieve your goals?

Here are some general tips that can help you stay focused and on track.

Set daily goals for studying

Ask yourself what you hope to accomplish in your course each day. Setting a clear goal can help you stay motivated and beat procrastination. The goal should be specific and easy to measure, such as "I'll watch all the videos in Module 2 and complete the first programming assignment". And don't forget to reward yourself when you make progress towards your goal!

Create a dedicated study space

It's easier to recall information if you're in the same place where you first learned it, so having a dedicated space at home to take online courses can make your learning more effective. Remove any distractions from the space and if possible, make it separate from your bed or sofa. A clear distinction between where you study and where you take breaks can help you focus.

Schedule time to study on your calendar

Open your calendar and choose a predictable, reliable time that you can dedicate to watching lectures and completing assignments. This helps ensure that your courses won't become the last thing on your to-do list.

Tip: You can add deadlines for a Coursera course to your Google calendar, Apple calendar, or another calendar app.

Keep yourself accountable

Tell your friends about the courses you're taking, post achievements to your social media accounts or blog about your homework assignments. Having a community and support network of friends and family to cheer you on makes a difference!

Actively take notes

Taking notes can promote active thinking, boost comprehension and extend your attention span. It's a good strategy to internalize knowledge whether you're learning online or in the classroom. So, grab a notebook or find a digital app that works best for you and start synthesizing key points.

Tip: While watching a lecture on Coursera, you can click the 'Save Note' button below the video to save a screenshot to your course notes and add your own comments.

Join the discussion

Course discussion forums are a great place to ask questions about assignments, discuss topics, share resources and make friends. Our research shows that learners who participate in the discussion forums are 37% more likely to complete a course. So make a post today!

Do one thing at a time

Multitasking is less productive than focusing on a single task at a time. Researchers from Stanford University found that "People who are regularly bombarded with several streams of electronic information cannot pay attention, recall information or switch from one job to another as well as those who complete one task at a time." Stay focused on one thing at a time. You'll absorb more information and complete assignments with greater productivity and ease than if you were trying to do many things at once.

Take breaks

Resting your brain after learning is critical to high performance. If you find yourself working on a challenging problem without much progress for an hour, take a break. Walking outside, taking a shower or talking with a friend can help you to re-energize and even give you new ideas on how to tackle the project.

Your learning journey starts now!

While preparing for the module quiz or working on achieving your learning goals you're encouraged to:

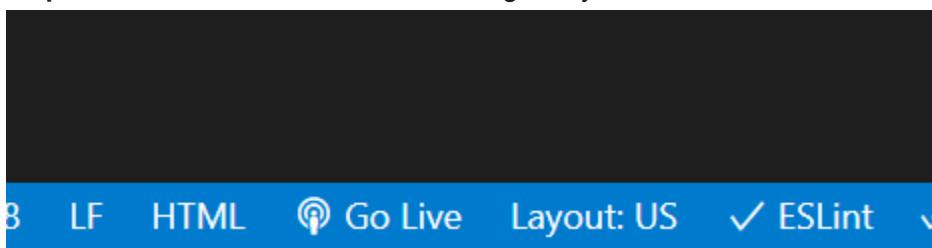
- Work through each lesson in the learning pathway. Try not to skip any activities or lessons unless you are certain that you already know this information well enough to move ahead.
- Take the opportunity to go back and watch a video or read all the information provided before moving on to the next lesson or module.
- Complete all the knowledge and module quizzes and exercises.
- Read the feedback carefully when answering quizzes, as this will help you to reinforce what you are learning.
- Make use of the practical learning environment provided by the exercises. You can gain substantial reinforcement of your learning through the step-by-step application of your skills.

Completed

Working with labs in this course

Throughout this course, you will be completing labs in a virtual coding environment. The purpose of these labs is to give you an opportunity to apply what you learn. You will be doing things like creating a basic login form for a website, a table booking form and a web page for rating a video. And that's just in the first module! In module 2 you'll create a grid layout and a restaurant menu and you'll practice the transform and transition properties in CSS. And, in the final module, there is a lab that will allow you to work in VS Code for the portfolio project without downloading it to your computer. To complete some of the labs you need to be able to see your rendered HTML in a browser. To view the rendered HTML on a web page follow the step-by-step instructions below.

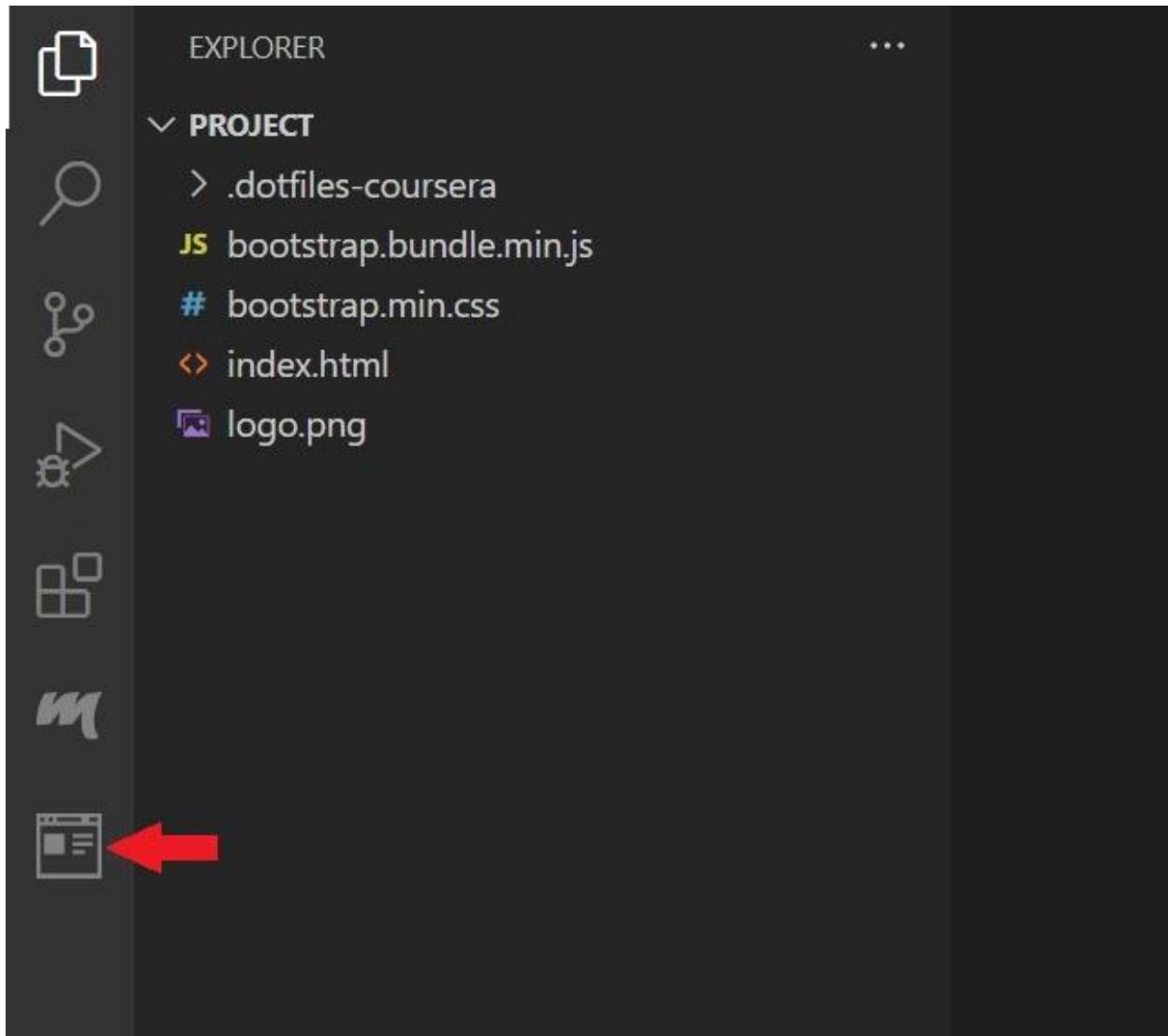
Step 1: Save the file.
Step 2: Click on 'Go live' at the bottom right of your editor.



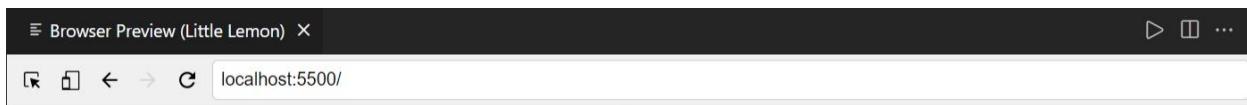
Once the server is up and running you'll see the exposed port.



Step 3: Click on browser preview.



Step 4: Enter the url as `http://localhost:5500`

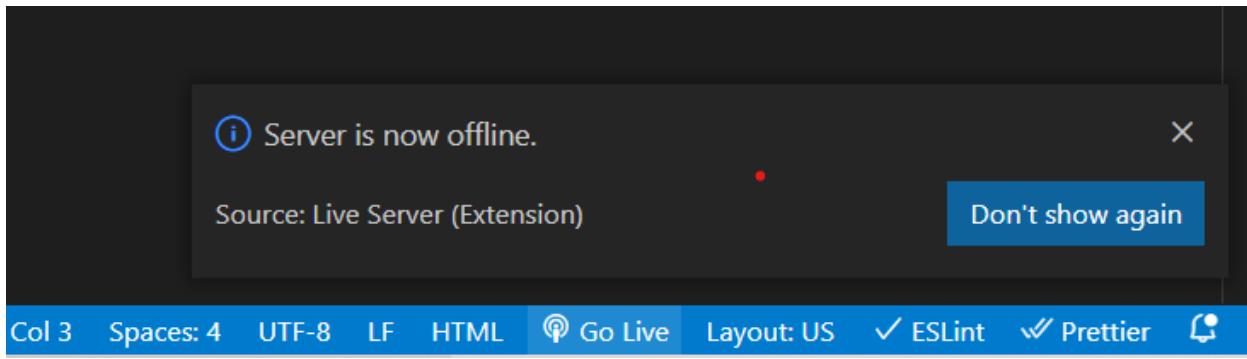


Step 5: Check that the web page displays.

After completing your lab, don't forget to close the server. You can close the server by clicking on the exposed Port number (e.g. 5500) after completing the lab.



You should see a notification like this which confirms the server has been stopped.



Completed

Semantic HTML cheat sheet

There are hundreds of semantic tags available to help describe the meaning of your HTML documents. Below is a cheat sheet with some of the most common ones you'll use in this course and in your development career.

Sectioning tags

Use the following tags to organize your HTML document into structured sections. **<header>** The header of a content section or the web page. The web page header often contains the website branding or logo. **<nav>** The navigation links of a section or the web page. **<footer>** The footer of a content section or the web page. On a web page, it often contains secondary links, the copyright notice, privacy policy and cookie policy links. **<main>** Specifies the main content of a section or the web page. **<aside>** A secondary set of content that is not required to understand the main content. **<article>** An independent, self-contained block of content such as a blog post or product. **<section>** A standalone section of the document that is often used within the body and article elements. **<details>** A collapsed section of content that can be expanded if the user wishes to view it. **<summary>** Specifies the summary or caption of a **<details>** element. **<h1><h2><h3><h4><h5><h6>** Headings on the web page. **<h1>** indicates the most important heading whereas **<h6>** indicates the least important.

Content tags

<blockquote> Used to describe a quotation. **<dd>** Used to define a description for the preceding **<dt>** element. **<dl>** Used to define a description list. **<dt>** Used to describe terms inside **<dl>** elements. **<figcaption>** Defines a caption for a photo image. **<figure>** Applies markup to a photo image. **<hr>** Adds a horizontal line to the parent element. **** Used to define an item within a list.

<menu> A semantic alternative to **** tag. **** Defines an ordered list. **<p>** Defines a paragraph.
<pre> Used to represent preformatted text. Typically rendered in the web browser using a monospace font. **** Unordered list

Inline tags

<a> An anchor link to another HTML document. **<abbr>** Specifies that the containing text is an abbreviation or acronym. **** Boldes the containing text. When used to indicate importance use **** instead. **
** A line break. Moves the subsequent text to a new line. **<cite>** Defines the title of creative work (for example a book, poem, song, movie, painting or sculpture). The text in the **<cite>** element is usually rendered in italics. **<code>** Indicates that the containing text is a block of computer code. **<data>** Indicates machine-readable data. **** Emphasizes the containing text. **<i>** The containing text is displayed in italics. Used to indicate idiomatic text or technical terms. **<mark>** The containing text should be marked or highlighted. **<q>** The containing text is a short quotation. **<s>** Displays the containing text with a strikethrough or line through it. **<samp>** The containing text represents a sample. **<small>** Used to represent small text, such as copyright and legal text. **** A generic element for grouping content for CSS styling. **** Displays the containing text in bold. Used to indicate importance. **<sub>** The containing text is subscript text, displayed with a lowered baseline. **<sup>** The containing text is superscript text, displayed with a raised baseline. **<time>** A semantic tag used to display both dates and times. **<u>** Displays the containing text with a solid underline. **<var>** The containing text is a variable in a mathematical expression.

Embedded content and media tags

<audio> Used to embed audio in web pages. **<canvas>** Used to render 2D and 3D graphics on web pages. **<embed>** Used as a containing element for external content provided by an external application such as a media player or plug-in application. **<iframe>** Used to embed a nested web page. **** Embeds an image on a web page. **<object>** Similar to **<embed>** but the content is provided by a web browser plug-in. **<picture>** An element that contains one **** element and one or more **<source>** elements to offer alternative images for different displays/devices. **<video>** Embeds a video on a web page. **<source>** Specifies media resources for **<picture>**, **<audio>** and **<video>** elements. **<svg>** Used to define Scalable Vector Graphics within a web page.

Table tags

<table> Defines a table element to display table data within a web page.
<thead> Represents the header content of a table. Typically contains one **<tr>** element.
<tbody> Represents the main content of a table. Contains one or more **<tr>** elements.
<tfoot> Represents the footer content of a table. Typically contains one **<tr>** element. **<tr>** Represents a row in a table. Contains one or more **<td>** elements when used within **<tbody>** or **<tfoot>**. When used within **<thead>**, contains one or more **<th>** elements. **<td>** Represents a cell in a table. Contains the text content of the cell. **<th>** Defines a header cell of a table. Contains the text content of the header. **<caption>** Defines the caption of a table element. **<colgroup>** Defines

a semantic group of one or more columns in a table for formatting. `<col>` Defines a semantic column in a table.

Completed

Metadata cheat sheet

HTML `<meta>` tags

Earlier in the course, you learned about meta tags and how you can leverage them to convey information to search engines to better categorize your pages. We recommend that you keep this cheat sheet handy when building your web applications. The structure of a meta tag is as follows.

Name The name of the property can be anything you like, although browsers usually expect a value they understand and can take an action upon. An example would be `<meta name="author" content="name">` to state the author of the page. **Content** The content field specifies the property's value. For example, you can use `<meta name="language" content="english">`, to specify the language of the webpage to search engines. **Charset** The charset is a special field that lets you specify the character encoding used for the page so that the browser can display it properly. The most frequently used is utf-8, and you would add it to your HTML header as follows: `<meta charset="UTF-8">` **HTTP-equiv** This field stands for HTTP equivalent, and it's used to simulate HTTP response headers. This is rare to see, and it's recommended to use HTTP headers over HTML http-equiv meta tags. For example, the next tag would instruct the browser to refresh the page every 30 minutes: `<meta http-equiv="refresh" content="30">`

Basic meta tags (meta tags For SEO)

`<meta name="description">` provides a brief description of the web page `<meta name="title">` specifies the title of the web page `<meta name="author" content="name">` specifies the author of the web page `<meta name="language" content="english">` specifies the language of the web page `<meta name="robots" content="index,follow" />` tells search engines how to crawl or index a certain page `<meta name="google">` tells Google not to show the sitelinks search box for your page when showing search results `<meta name="googlebot" content="notranslate" />` tells Google you don't want to provide an automatic translation for your page if the user uses a different language `<meta name="revised" content="Sunday, July 18th, 2010, 5:15 pm" />` specifies the last modified date and time on which you have made certain changes `<meta name="rating" content="safe for kids">` specifies the expected audience for your page `<meta name="copyright" content="Copyright 2022">` specifies a Copyright

`<meta http-equiv="...">` tags

`<meta http-equiv="content-type" content="text/html">` specifies the format of the document returned by the server
`<meta http-equiv="default-style"/>` specifies the format of the styling document
`<meta http-equiv="refresh"/>` specifies the duration of the page before it's considered stale
`<meta http-equiv="Content-language"/>` specifies the language of the page
`<meta http-equiv="Cache-Control" content="no-cache">` instructs the browser how to cache your page

Responsive design/mobile meta tags

`<meta name="format-detection" content="telephone=yes"/>` indicates that telephone numbers should appear as hypertext links that can be clicked to make a phone call
`<meta name="HandheldFriendly" content="true"/>` specifies that the page can be properly visualized on mobile devices
`<meta name="viewport" content="width=device-width, initial-scale=1.0"/>` specifies the area of the window in which web content can be seen
Completed

Layout design

As you build web pages throughout your career, you'll notice that many pages follow similar layouts and structures. This is the outcome of many years of research into user interface design and user experience. Different companies, libraries and frameworks then adopt the resulting best practices. Many examples of these layouts can be seen in the popular bootstrap framework. However, many other frameworks provide similar designs.

Top navbar layout

Websites often have a top navbar layout to provide a set of essential anchor links to the user. These typically link to the main areas of the website, such as product pages, careers pages or contact pages. This provides the visitor to the website with a consistent navigation experience.

Navbar example

This example is a quick exercise to illustrate how the top-aligned navbar works. As you scroll, this navbar remains in its original position and moves with the rest of the page.

[View navbar docs »](#)

Carousel layout

Product-focused websites often use a large carousel on their homepage to highlight their featured products, discounts and offers. The carousel contains content items that will rotate through the carousel area at a fixed interval.



Example headline.

Some representative placeholder content for the first slide of the carousel.

[Sign up today](#)



Blog layout

The blog layout is used to feature multiple content items of differing importance. It is often seen on news websites where new articles will appear on the page each day based on current events.

[Subscribe](#)

Large

[Sign up](#)[World](#) [U.S.](#) [Technology](#) [Design](#) [Culture](#) [Business](#) [Politics](#) [Opinion](#) [Science](#) [Health](#) [Style](#) [Travel](#)

Title of a longer featured blog post

Multiple lines of text that form the lede, informing new readers quickly and efficiently about what's most interesting in this post's contents.

[Continue reading...](#)[World](#)

Featured post

Nov 12

This is a wider card with

Thumbnail

[Design](#)

Post title

Nov 11

This is a wider card with

Thumbnail

The layout typically features different-sized feature areas followed by a series of article summary areas that link to full articles.

[World](#)

Featured post

Nov 12

This is a wider card with supporting text below as a natural lead-in to additional content.

[Continue reading](#)

Thumbnail

[Design](#)

Post title

Nov 11

This is a wider card with supporting text below as a natural lead-in to additional content.

[Continue reading](#)

Thumbnail

From the Firehose

Sample blog post

January 1, 2021 by [Mark](#)

This blog post shows a few different types of content that's supported and styled with Bootstrap. Basic typography, lists, tables, images, code, and more are all supported as expected.

This is some additional paragraph placeholder content. It has been written to fill the

About

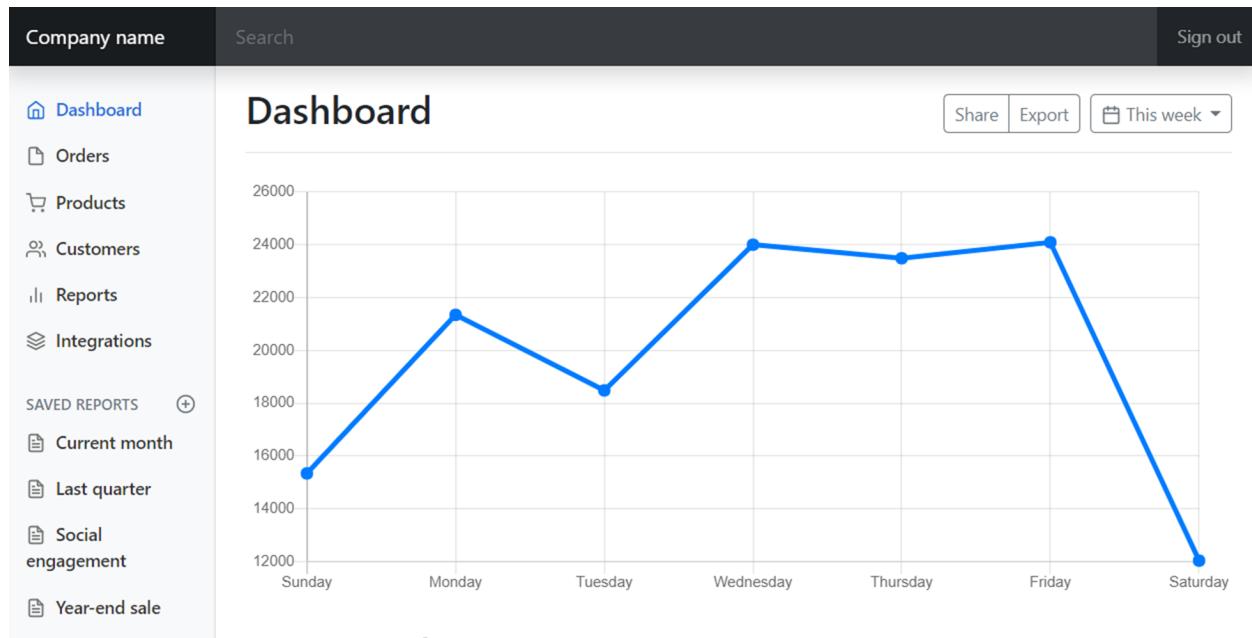
Customize this section to tell your visitors a little bit about your publication, writers, content, or something else entirely. Totally up to you.

Archives

[March 2021](#)

Dashboard layout

Dashboard layouts are often used in enterprise software for managing various web applications. They typically feature a sidebar for navigation with the main content area containing forms for configuration or reporting data such as graphs and tables. This trendy layout provides a good user experience for business users.



More layouts

You can explore more of these layouts on the bootstrap examples page in the additional resources. Consider these layouts when building websites and web applications so that you provide your audience with the best user experience possible.

Completed

Social media cards

Before Facebook introduced the Open Graph Protocol (OGP), search engine web crawlers, including social media websites, used the internal heuristics of a website to make the best possible guess in terms of the title, description, and preview images to be used for the content. This often led to social networks not completely successfully interpreting the post or information shared in the URL provided. Even today sometimes when you open a link for a website, the preview generated will be an out-of-scale image or a random image that is possibly embedded somewhere within the web page. This is where meta tags come in to help the end-user take better control of their content.

Over the years, the successful implementation of OGP has led to other social media giants, including sister companies owned by Meta, to adopt the protocol to improve the user experience. These platforms have their own meta tags that prefix and replace 'og' that you have encountered earlier in the course.

Need for social media cards

With the number of users and the use of internet marketing on the rise, user attention is the currency. It is said that a picture is worth a thousand words and the internet is a living proof of it: a caption and image can drive users towards or away from a website. The title and description shared with an URL should summarize the contents of a web page. In some cases, it may refer to generic information about the entire website. While for others you may tailor the social media (SM) card for a specific page on a website that you're sharing. 'type' is an important OGP tag in SM cards that help describe the details of a link such as if it's a book, article, movie and so on, and provide more detailed metadata for specific types. For example, in case of the music type, you can add details for the song, album, duration or any other information about the song. For a regular user scrolling through social media, the link provided with the image preview has only one chance and a moment's attention to make a good impression. In such cases, the role of social media cards becomes very important.

The extra time spent by a developer to add social media tags is worth the effort!

Social media cards and SEO

The internet today is an interconnected graph that is internally a web of URL links, web crawlers, and search engine optimization tools. Together, a web page's image and title are the store front to invite the user. But the social media cards also play an important role in boosting the rankings for the web crawlers used by search engines. They provide the crawlers with the necessary information to build metadata that eventually helps in ranking websites. Additionally, it also helps track traffic to your website.

While it's advised to stick to the required tags in social media cards, a developer can also use auxiliary tags that may be suitable. For example, the use of the video tag that helps to play in-line when displayed on social media websites like Meta.

Meta also has a dedicated page to assist developers that you can find in the additional resources for this section.

Completed

Additional resources

The following resources will be helpful as additional references in dealing with different concepts related to the topics you have covered in this section.

[HTML meta tags](#)

[Semantic elements](#)

[Simple bare bones HTML webpage](#)

[HTML5/CSS bare-bones newsletter template](#)

[Add open graph social metadata- Twitter](#)
[Essential meta tags for social media](#)
[The meta element](#)
[Open graph protocol](#)
[Using open graph protocol on website](#)
[Meta OGP guide for webmasters](#)
[Bootstrap with HTML](#)
[Bootstrap Layout Examples](#)
Completed

Input types

You already learned about the input HTML tag and how the type property determines the data your users can type in. This cheat sheet should be a reference to decide what type best suits your use case. Most of the inputs go hand in hand with the label tag for best accessibility practices.

Button

This displays a clickable button and it's mostly used in HTML forms to activate a script when clicked.

`<input type="button" value="Click me" onclick="msg()" />`

Keep in mind you can also define buttons with the `<button>` tag, with the added benefit of being able to place content like text or images inside the tag.

1

2

3

4

```
<button onclick="alert('Are you sure you want to continue?')">  
    
</button>
```

Checkbox

Defines a check box allowing single values to be selected or deselected. They are used to let a user select one or more options of a limited number of choices.

1

2

3

4

```
<input type="checkbox" id="dog" name="dog" value="Dog">  
  
<label for="dog">I like dogs</label>  
  
<input type="checkbox" id="cat" name="cat" value="Cat">  
  
<label for="cat">I like cats</label>
```

Radio

Displays a radio button, allowing only a single value to be selected out of multiple choices. They are normally presented in radio groups, which is a collection of radio buttons describing a set of related options that share the same "name" attribute.

1

2

3

4

```
<input type="radio" id="light" name="theme" value="Light">  
  
<label for="light">Light</label>  
  
<input type="radio" id="dark" name="theme" value="Dark">  
  
<label for="dark">Dark</label>
```

Submit

Displays a submit button for submitting all values from an HTML form to a form-handler, typically a server. The form-handler is specified in the form's "action" attribute:

```
1 <form action="myserver.com" method="POST">  
  
2   ...  
  
3   <input type="submit" value="Submit" />  
  
4 </form>
```

Text

Defines a basic single-line text field that a user can enter text into.

```
1 <label for="fname">First name:</label>  
  
2 <input type="text" id="fname" name="fname">
```

Password

Defines a single-line text field whose value is obscured, suited for sensitive information like passwords.

```
1 <label for="pwd">Password:</label>  
  
2 <input type="password" id="pwd" name="pwd">
```

Date

Displays a control for entering a date with no time (year, month and day).

1

2

```
<label for="dob">Date of birth:</label>

<input type="date" id="dob" name="date of birth">
```

Datetime-local

Defines a control for entering a date and time, including the year, month and day, as well as the time in hours and minutes.

1

2

```
<label for="birthdaytime">Birthday (date and time) :</label>

<input type="datetime-local" id="birthdaytime" name="birthdaytime">
```

Email

Defines a field for an email address. It's similar to a plain text input, with the addition that it validates automatically when submitted to the server.

1

2

```
<label for="email">Enter your email:</label>

<input type="email" id="email" name="email">
```

File

Displays a control that lets the user select and upload a file from their computer. To define the types of files permissible you can use the "accept" attribute. Also, to enable multiple files to be selected, add the "multiple" attribute.

1

2

```
<label for="myfile">Select a file:</label>
```

```
<input type="file" id="myfile" name="myfile">
```

Hidden

Defines a control that is not displayed but whose value is still submitted to the server.

1

```
<input type="hidden" id="custId" name="custId" value="3487">
```

Image

Defines an image as a graphical submit button. You should use the “src” attribute to point to the location of your image file.

1

```
<input type="image" src="submit_img.png" alt="Submit" width="48" height="48">
```

Number

Defines a control for entering a number. You can use attributes to specify restrictions, such as min and max values allowed, number intervals or a default value.

1

```
<input type="number" id="quantity" name="quantity" min="1" max="5">
```

Range

Displays a range widget for specifying a number between two values. The precise value, however, is not considered important. This is typically represented using a slider or dial control. To define the range of acceptable values, use the “min” and “max” properties.

1

2

```
<label for="volume">Volume:</label>
```

```
<input type="range" id="volume" name="volume" min="0" max="10">
```

Reset

Displays a button that resets the contents of the form to their default values.

1

```
<input type="reset">
```

Search

Defines a text field for entering a search query. These are functionally identical to text inputs, but may be styled differently depending on the browser.

1

2

```
<label for="gsearch">Search in Google:</label>  
  
<input type="search" id="gsearch" name="gsearch">
```

Time

Displays a control for entering a time value in hours and minutes, with no time zone.

1

2

```
<label for="appt">Select a time:</label>  
  
<input type="time" id="appt" name="appt">
```

Tel

Defines a control for entering a telephone number. Browsers that do not support "tel" fall back to standard text input. You can optionally use the "pattern" field to perform validation.

1

2

```
<label for="phone">Enter your phone number:</label>  
  
<input type="tel" id="phone" name="phone" pattern="[\+]{1}[0-9]{11,14}">
```

Url

Displays a field for entering a text URL. It works similar to a text input, but performs automatic validation before being submitted to the server.

1

```
<label for="homepage">Add your homepage:</label>

<input type="url" id="homepage" name="homepage">
```

Week

Defines a control for entering a date consisting of a week-year number and a year, with no time zone. Keep in mind that this is a newer type that is not supported by all the browsers.

1

2

```
<label for="week">Select a week:</label>

<input type="week" id="week" name="week">
```

Month

Displays a control for entering a month and year, with no time zone. Keep in mind that this is a newer type that is not supported by all the browsers.

1

2

```
<label for="bdaymonth">Birthday (month and year) :</label>

<input type="month" id="bdaymonth" name="bdaymonth" min="1930-01"
value="2000-01">
```

Completed

Visual Studio Code on Coursera

In addition to having Visual Studio Code installed on your own computer, in this course and throughout this program, you'll have the opportunity to work in Visual Studio Code right here on Coursera!

As you progress through the course, you'll be able to write code in hands-on activities called **Labs**. In these labs you'll be able to open Visual Studio Code and start writing code without ever leaving the course.

Lab: Creating an HTML Document

How to get started working on Labs

The Labs contain instructions explaining the coding task.

Creating an HTML Document

[Open Lab](#) ↗

Instructions



Introduction

In this ungraded lab you will practice creating a simple HTML document.

Goal

- Create a valid HTML document that displays a piece of text.

Objectives

- Add the DOCTYPE.
- Add the HTML, head and body elements.
- Add the title element.
- Add the text to the body element.

Instructions

When you click the button to open the lab, a new tab will open with Visual Studio Code already setup and ready for you to start writing code!



☰ Navigate ⚒ Lab Files ⓘ Help

The screenshot shows the Visual Studio Code interface. The top bar includes the Coursera logo, navigation links (☰ Navigate, ⚒ Lab Files, ⓘ Help), and file tabs (index.html — project — code-server). The left sidebar (Explorer) shows a project structure with a folder named ".dotfiles-coursera" containing an "index.html" file. The main editor area displays the content of "index.html". The bottom status bar shows connection details (wdblkcv.a.labs.coursera.org), code analysis results (0△0), and various development settings like Open Development Server, Layout: U.S., ESLint, and Prettier.

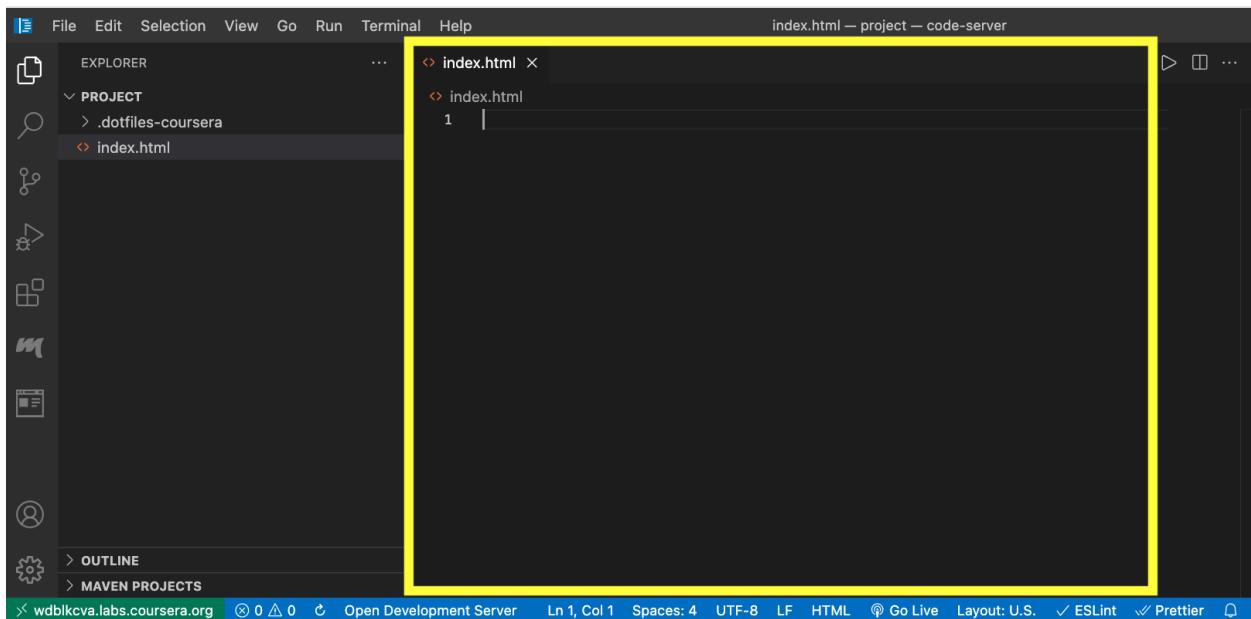
You'll see all the files for the lab in the Project folder in the left sidebar.



☰ Navigate ⚒ Lab Files ⓘ Help

This screenshot is identical to the one above, but the Explorer sidebar on the left is highlighted with a thick yellow border. The rest of the interface, including the editor area and status bar, remains the same.

And the large editor area where you write your code for the lab.



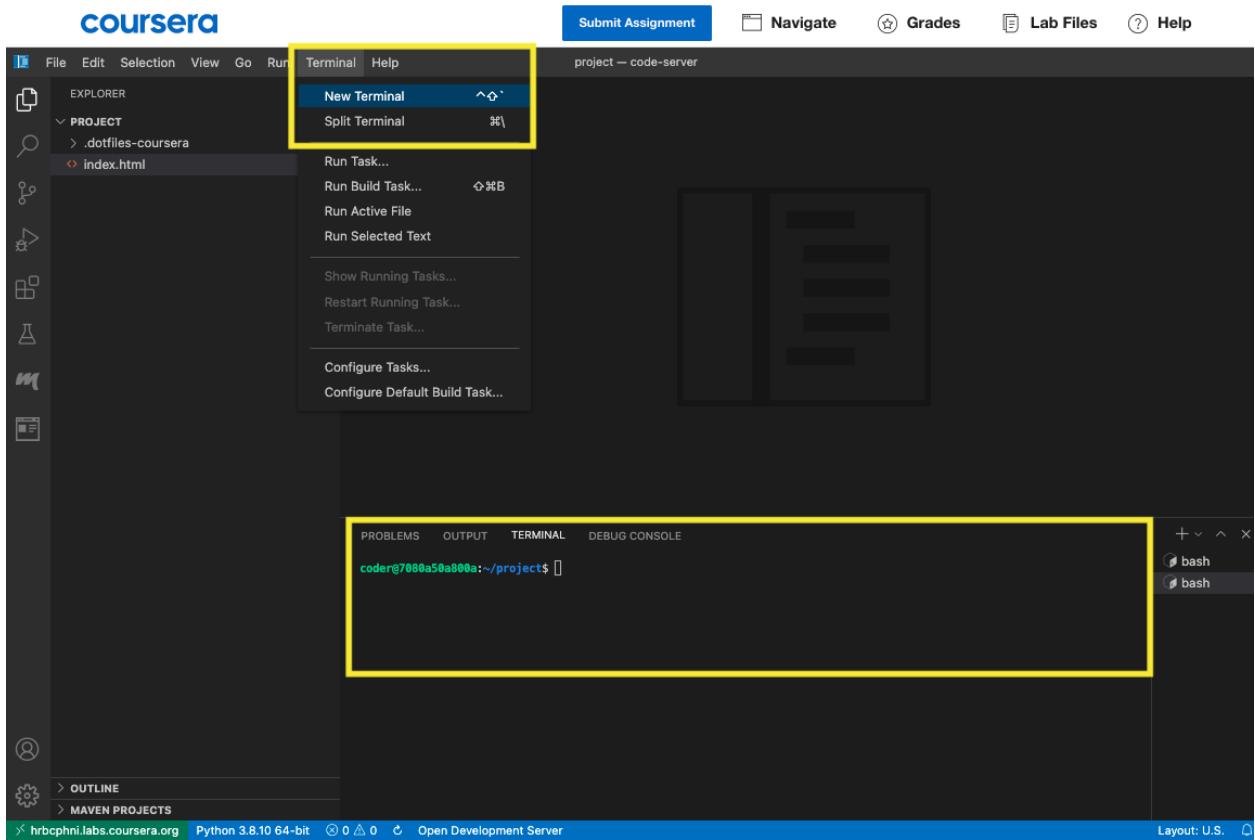
All **Lab** exercises will include two important files. A **README.md** file which contains instructions for using Visual Studio Code and most importantly, how to run and view the output of your code. The **README.md** file is the same for every Lab. There will also be an **instructions.md** file which provides specific instructions for each Lab.

These files may look a little confusing at first. They're written using a language called Markdown. Markdown is just used to add formatting to text elements. Don't worry, you don't have to know anything about Markdown. If you open these files in **Preview** you'll just see the formatted output and they'll look totally normal.

You'll also likely see quite a few files and folders when you open in the Lab. Many of the files and folders you'll be able to ignore. To get started, look at the **instructions.md** file and it will explain which file or files you'll be working in.

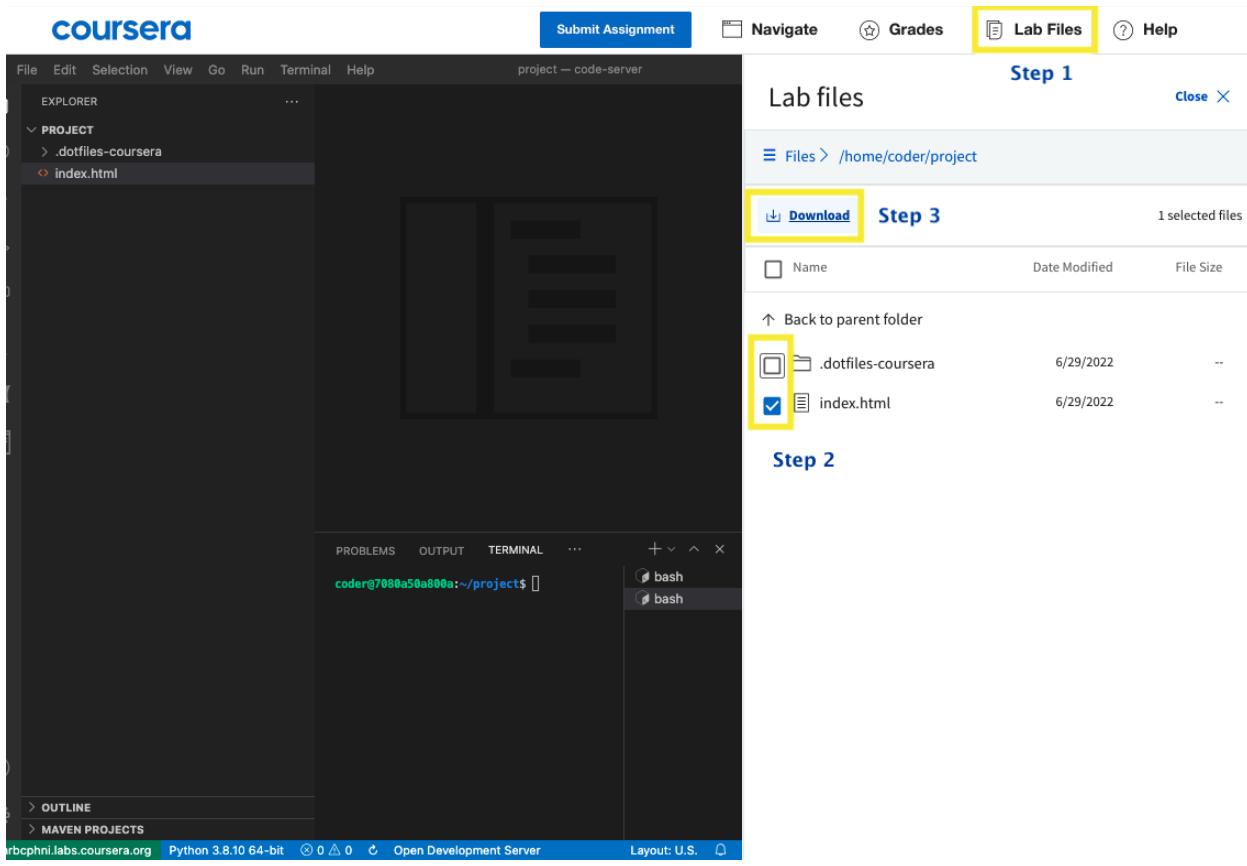
Working in the Terminal

For Labs in some courses, you may need to use a tool called the Terminal from time to time to complete course activities. You can open this by selecting the **Terminal** option in the upper Visual Studio Code toolbar.



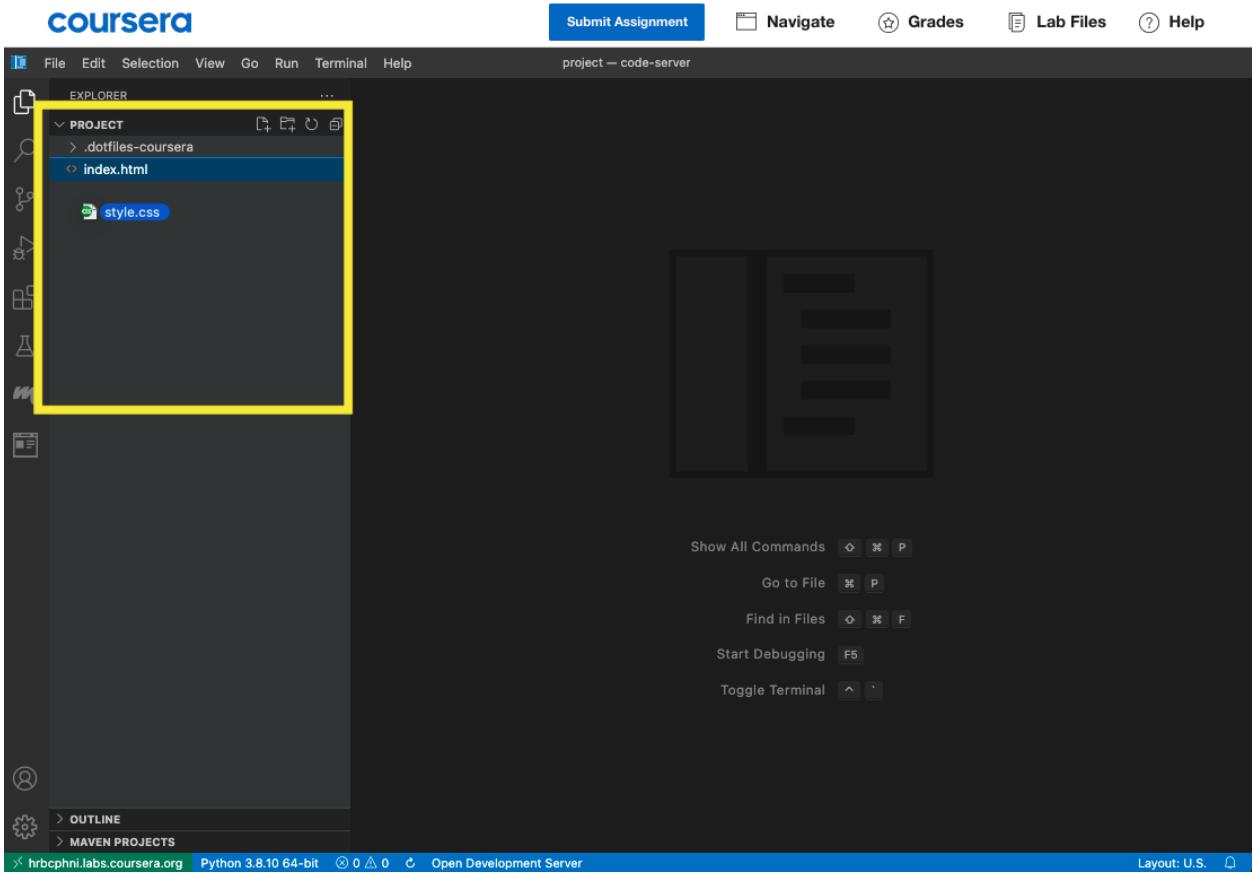
How to download files from your Visual Studio Code Lab to your local device

1. Select the **Lab Files** button in your Lab Toolbar.
2. You'll be able to download your full workspace, specific folders, or individual files through the checkbox selection tool.
3. After you've selected these files, use the **Download** link to download your files to your local device.



How to upload local files to your Visual Studio Code Lab

If you'd like to upload your course files from your local device to your Visual Studio Code lab, **drag and drop** your file from your local device into the Visual Studio Code file tree.



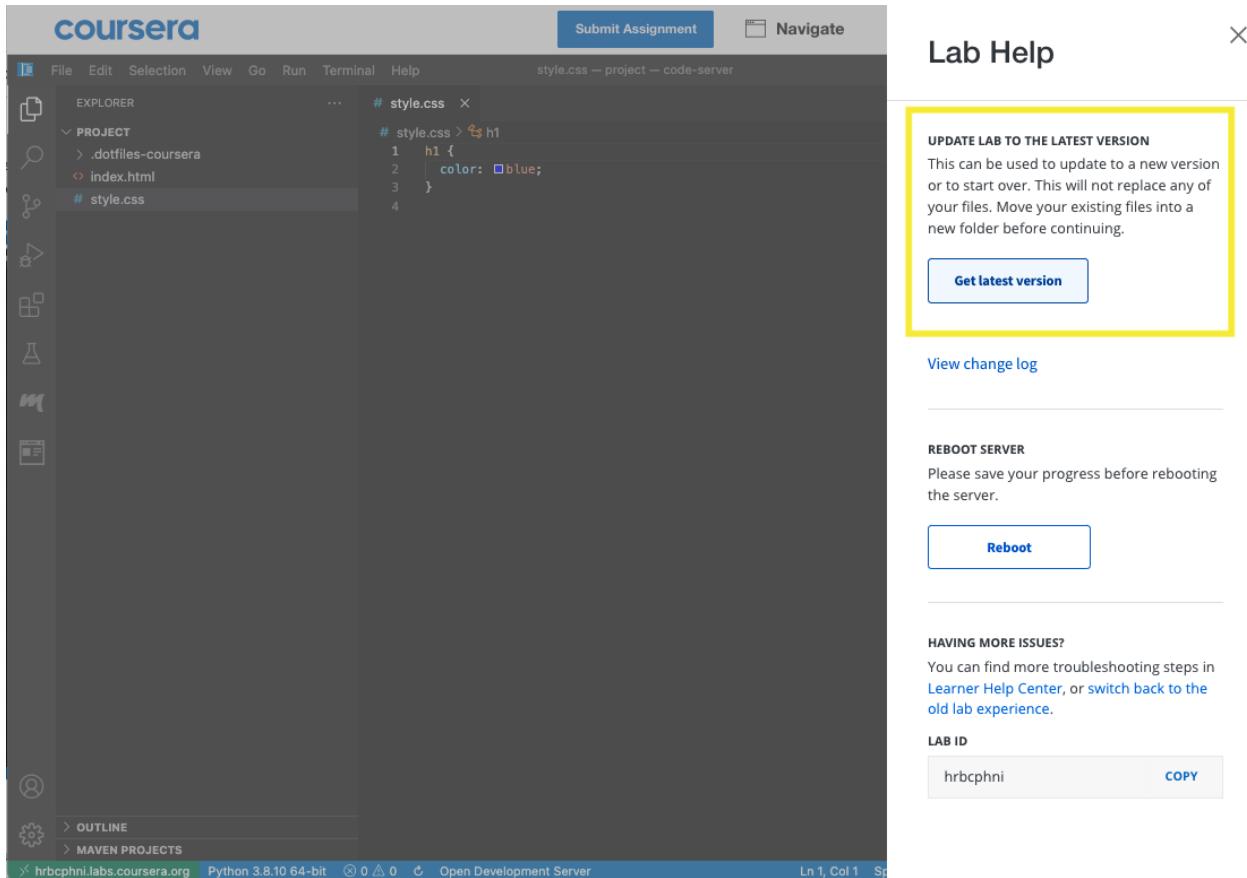
How to get a fresh copy of course-provided starter files

Your work will be saved and persist within your Visual Studio Code lab while you are enrolled in the course. If you'd like to get a fresh copy of the original instructor-provided files at any time, you can do this through the **Lab Help** option in your Lab Toolbar. Don't worry - your original work and files will still remain in your lab until you personally remove or delete them, even when refreshing your files through the steps below.

1. First rename your original files to something like `[yourfilename] [original].[your file extension]`. You can do this by right-clicking on your file in the Visual Studio Code file tree, selecting **Rename**, and providing a new file name.

- For example for `index.html`, this could be renamed to `'index [original].html'`

2. Select **Lab Help** from your Lab Toolbar and then select **Get latest version**.



3. You should now see a fresh copy of the original instructor-provided files in your lab, in addition to your own (renamed) files.

Completed

Create and test a form (solution)

The following code is an example solution for the previous exercise.
This code is placed inside the `<main>` element of `index.html`.

1

2

```
3  
4  
5  
6  
7  
8  
9  
10  
11  
  
<form>  
  
  <div>  
    <label for="username">Username</label>  
  
    <input type="text" id="username" required minlength="2">  
  
  </div>  
  
  <div>  
    <label for="password">Password</label>  
  
    <input type="password" id="password" required minlength="2">  
  
  </div>  
  
  <button type="submit">Log In</button>  
  
</form>
```

While reviewing the code, note the following items:

- The label *for* attribute value matches the value of the *id* attribute on the corresponding input element
- The *type* attribute is set to *text* for the username input
- The *type* attribute is set to *password* for the password input
- Each field has a *required* attribute to enable client-side validation which checks that the form is filled out by the user
- Each field has a *minlength* attribute with its value set to 2. This enables client-side validation which will prompt the user if the content of the field is less than 2 characters.

Completed

Cheat sheet: Interactive form elements

When filling in HTML forms, we expect users to abide by certain rules, like using numbers when asked to, or properly formatting a URL or an email when needed. However, humans are prone to errors and in some cases, they may overlook some of the data they input. That's why it's important to ensure the shape of the data we expect in each field is correct. HTML form validation is a set of attributes we can add to form inputs to perform automatic validation on the user's behalf. The most important attributes you'll find yourself using for validation are the following.

Required

Denotes a mandatory input that the user can't leave empty. It can be used with any input type, like password, radio, text and so on. `<input type="text" id="firstName" name="firstName" required>`

Maxlength

Specifies the maximum length of a text input, in other words, the maximum number of characters that can be entered for a specific field. If provided, it will prevent the user from entering more characters than the limit. `<input type="text" id="description" name="description" maxlength="50">`

Minlength

Specifies the minimum length of a text input. If set, the input will not accept fewer characters than those specified. `<input type="password" id="password" name="password" minlength="8">`

Min and max attributes

Determine the minimum and maximum values allowed for an input field. They are usually applied to numerical text inputs, range inputs or dates.

```
<input type="number" id="quantity" name="quantity" min="1" max="10"><input type="range" id="volume" name="volume" min="1" max="100">
```

Multiple

Indicates that the user can enter more than one value in a single input field. This attribute can only be used for email and file input types.

```
<input type="file" id="gallery" name="gallery" multiple>
```

Pattern

Defines a particular pattern that an input field value has to fulfill to be considered valid. This attribute expects a regular expression to specify the pattern. It works with text, date, search, URL, tel, email and password input types. For example, you can restrict phone numbers to be only from the UK.

```
<input type="tel" id="phone" name="phone" pattern="^(?:0|+\d{10})$>
```

Completed

Submit

You have recently learned about how forms are sent to web servers and the difference between Get and Post. In this reading, you will build on this knowledge by learning about Submit.

Action and method

Form submissions are an essential part of the world wide web. Nearly every website uses forms, from buying items online to ordering food for delivery. When you click the login button on a website, it sends your username and password to a web server to log you into your account. As you know by now, you add a form to your web page using the form tag.

1

2

```
<form>
```

```
</form>
```

But how the form is submitted is determined by two essential attributes: action and method. The action attribute specifies to which web address the form must be sent. This is address is location of server-side code that will process the request.

[1](#)

[2](#)

```
<form action="/login">  
</form>
```

It is important to note that action can be a full URL address such as `https://meta.com`, an absolute path such as `/login`, or a relative path such as `login`. The absolute path, which starts with a forward slash, will use the base address of the current website, such as `https://meta.com` and combine it with the absolute path. For example, if `/login` is the absolute path, the form will be submitted to `https://meta.com/login`. If the address is `https://meta.com/company-info/` and `/login` is the absolute path, the submission address will still be `https://meta.com/login`. Similarly, a relative path will combine the current web address with a relative path. For example, if the web browser is currently on the web page `https://meta.com/company-info/`, and the relative path is set to `login`, the form will be submitted to

`https://meta.com/company-info/login`. The method attribute specifies which HTTP method is used to submit the form; GET or POST.

[1](#)

[2](#)

```
<form method="get">  
</form>
```

[1](#)

[2](#)

```
<form method="post">  
</form>
```

The form will default to the HTTP GET method when the method attribute is not provided. As you may already know, when the form is submitted using the HTTP GET method, the data in the form's fields are encoded in the URL. And when the form is submitted using the HTTP POST method, the data is sent as part of the HTTP request body. When the web server receives the request, it processes the data and sends back an HTTP response. The response indicates the result of the

submission, which can be successful or fail due to invalid or incorrect data. However, as a front-end developer, it is essential to know that forms are not the only way to submit data to the web server. As you learn more about JavaScript and front-end libraries, you'll discover that developers often submit HTTP requests directly via code and send data as part of the HTTP request body in a text format called JavaScript Object Notation, or JSON. But that is a topic for another course. For now, practice building HTML forms and submitting data to the web server using the different attributes available.

Completed

Glossary: HTML form elements

The `<form>` element in HTML is an important and useful element. The following sheet provides an overview of the `<form>` constituent elements and their common attributes with simple examples for quick reference.

`<input>`

It is used to create interactive controls, for example, buttons and various types of text fields and so on, to accept input or data from the user. The key attribute of this element is `type`. Some common values for the `type` include: `button`, `checkbox`, `date`, `email`, `number`, `password`, `submit`, `text`, and `url`. These values dictate the appearance of the element. For example, this code:

1

2

3

4

5

6

7

8

9

10

11

12

```
<form action="my_action_page">

<label for="uname">Username:</label>

<br>

<input type="text" id="uname" name="username">

<br>

<label for="pwd">Password:</label>

<br>

<input type="password" id="pwd" name="pwd">

<br><br>

<input type="submit" value="Login">

</form>
```

Results in the following output:

Username:

Rita

Password:

.....

Login

Note how the type `password` hides the user input.

<label>

Defines a label for an element. It has an attribute "for", the value of which should be equal to the id attribute of the element it is associated with. Note how in the example above, the <label> is associated with the <input> using its id value.

<select>

Defines a drop-down list of options presented to the user. It has a couple of attributes:

- Form, the id of the form in which the drop-down appears
- Name specifies the name of the control
- Multiple Boolean attribute, when specified, indicates if a user can select multiple options out of the list
- Required indicates if the user is required to select an option before submitting a form
- Size mentions the number of visible options in a drop-down list

The options in a drop-down list are defined using the <option> element inside <select>. Note the example in the <option> description below.

<textarea>

Defines a multi-line input field, typically to allow the user to input longer textual data. The common attributes for this element include:

- `cols` defines the width of the text area, the default value is 20
- `form` the form element the text area is associated with
- `maxlength` when specified, limits the maximum number of characters that can be entered in the text area

- `minlength` the minimum number of characters that the user should enter
- `readonly` once set, the user cannot modify the contents
- `rows` defines the number of visible text lines for the text area

The following line of code defines a text area of 10 visible lines and nearly 30 characters wide where the user can input a maximum of 200 characters:

```
1 <textarea name="response" rows="10" cols="30" maxlength="200">  
2  
3 </textarea>
```

<button>

Defines a clickable button. The `onclick` attribute defines the behavior when the button is clicked by the user. For example, in the code below, an alert message is shown to the user.

```
1 <button type="button" onclick="alert('You just clicked!')">Click Me!  
2  
3 </button>
```

<fieldset>

Used to group related input elements in a form. For instance, elements related to the user's personal information and educational qualification can be grouped separately in two field sets.

<legend>

Defines a caption for the `<fieldset>` element. For example:

```
1  
2  
3
```

```
4
5
6
7
8
9
10
11
12
13
14
15

<fieldset>

<legend>Personal Info</legend>

<label for="fname">First name:</label><br>

<input type="text" id="fname" name="fname" value="John"><br>

<label for="lname">Last name:</label><br>

<input type="text" id="lname" name="lname" value="Doe"><br>

</fieldset>

<fieldset>
```

```
<legend>Qualificaiton</legend>

<label for="pdegree">Primary degree:</label><br>

<input type="text" id="pdegree" name="degree" value="Masters"><br>

<label for="fos">Last name:</label><br>

<input type="text" id="fos" name="field" value="Psychology"><br>

</fieldset>
```

<datalist>

Specifies a list of pre-defined options for an input element. It differs from <select> since the user can still provide textual or numeric input other than the listed options.

1

2

3

4

5

6

7

8

9

10

11

12

```
<form action="/my_action_page">
```

```
<label for="flowers">Favourite flower:</label><br>

<input list="flowers" name="flowers">

<datalist id="flowers">

<option value="Rose">

<option value="Lily">

<option value="Tulip">

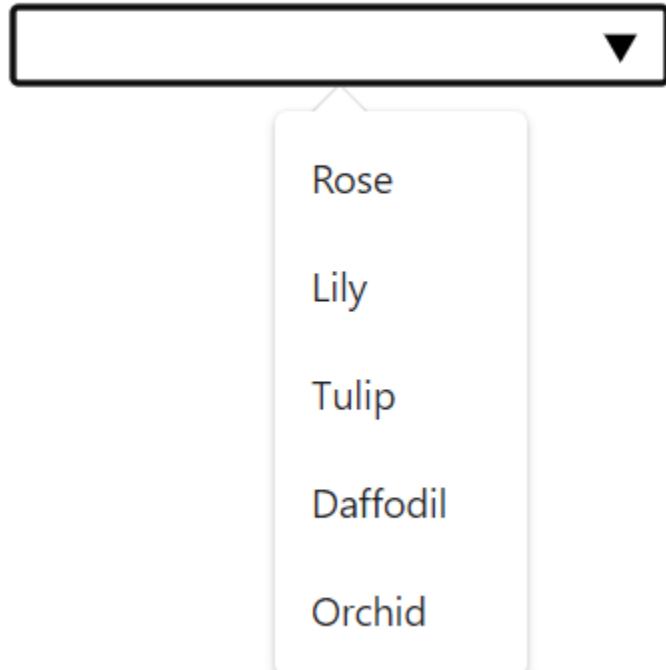
<option value="Daffodil">

<option value="Orchid">

</datalist>

...
</form>
```

Favourite flower:



<output>

Represents the result of a calculation (typically the output of a script) or the outcome of the user action.

<option>

Defines an option for the drop-down list. The following code example demonstrates how a simple list can be defined, with the rendered view below the code block.

```
1<label for="course">Choose a course:</label><br>
2<select id="course" name="courselist">
3    <option value="html">HTML Introduction</option>
4    <option value="css">Styling with CSS</option>
5    <option value="js">JavaScript</option>
6    <option value="react">React Basics</option>
7</select>
```

Choose a course:

HTML Introduction ▾

HTML Introduction

Styling with CSS

JavaScript

React Basics

By default, the first item in the drop-down list is selected. To define a pre-selected option, add the `selected` attribute to the option.

<optgroup>

Defines a group of related options in a drop-down list. Its attribute `label` names the group.
Completed

Create a complex form (solution)

The following code is an example solution for the previous exercise.
This code is placed inside the `<main>` element of `index.html`.

1

2

3

4

5

```
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

<form>

<div>

    <label for="email_address">Email Address</label>

    <input type="email" id="email_address" name="email_address">

</div>
```

```

<div>

    <label for="booking_date">Date of Booking</label>

    <input type="date" id="booking_date" name="booking_date">

</div>

<div>

    <label for="people">Number of people</label>

    <input type="number" id="people" min="1" max="8" name="people">

</div>

<div>

    <label>

        <input type="checkbox" id="agree" name="agree" required>

        I agree to the cancellation policy

    </label>

</div>

<button type="submit">Book Now</button>

</form>

```

While reviewing the code, note the following:

- The `type` attribute is set to `email` for the email address field. This will enable client-side validation to ensure that the user enters a correctly formatted email address.
- The `type` attribute is set to `date` for the booking date field. This will enable the browser's built-in date picker for the field.
- The `type` attribute is set to `number` for the number of people field. This will use the browser's built-in number picker for the field.
- The `min` attribute is added to the people input element to set the minimum value to 1.
- The `max` attribute is added to the people input element to set the maximum value to 8.

- The checkbox input element is contained inside the label element. For mobile devices, this will improve the user experience so that the user can touch either the checkbox or the text "I agree to the cancellation policy" to toggle the checkbox.
- The checkbox input element has a *required* attribute. This will require that the user agrees to the cancellation policy before being able to book a table.

Completed

Additional resources

The following resources will be helpful as additional references in dealing with different concepts related to the topics you have covered in this section.

[Client-side validation of forms with HTML5](#)

[<input> tag in HTML](#)

[Form validation examples](#)

[Input type: Radio buttons](#)

[Why does your website look different in different browsers?](#)

[HTML Form submission – sending form data](#)

Completed

Rate the media (solution)

The following is a sample solution to the previous exercise.

The code below is placed inside the <body> element of index.html.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

```
<video controls width="320" height="240">  
  <source src="video.mp4" type="video/mp4">  
</video>
```

```
<form>  
  <div>  
    <label for="rating">Rating</label>  
  
    <input type="range" min="1" max="5" id="rating" name="rating"  
          list="ratings">  
  
    <datalist id="ratings">
```

```

<option value="1" label="1"></option>

<option value="2"></option>

<option value="3"></option>

<option value="4"></option>

<option value="5" label="5"></option>

</datalist>

</div>

<button type="submit">Submit Rating</button>

</form>

```

While reviewing the code, note the following:

- The `video` element has a `controls` attribute to enable the video player controls (pause, volume, timeline, and so on).
- The `video` element contains a `source` element. The `source` element has a `src` attribute which specifies the video file to be played. It also has a `type` attribute to specify the video type, in this case it is an MP4 file.
- The input type for the rating is set to `range`. This will present a slider in the web browser. The `min` attribute is set to 1 and the `max` attribute is set to 5. The default range is 0 to 100 so these attributes need to be set so that the slider range matches the `datalist`.
- The range input has a `list` attribute set to ratings. This matches the `id` attribute of the `datalist` element below the input element.
- The `datalist` element contains the possible slider value and the corresponding label to display. When no label is specified for a value, no label will display on the slider for that value.

Completed

Images

This lesson will help refresh your knowledge of the `` tag and how you can use it to embed images in webpages. The `` tag is used to add an image to a web page. The image's address is specified using the `src` attribute. For example, if you wanted to embed an image file named `photo.png`, you can do that with the following HTML. `` You can also

specify the width and height of the image using the width and height attributes. For example, if the width of the photo is 640 pixels and the height of the photo is 480 pixels, you can use the following HTML. `` It is important to note that you can set the image to a larger or smaller size and the web browser will automatically scale the image. For example, you can update the previous HTML to half the width and height and the image would shrink by 50%. `` One useful feature of rendering images in the web browser is that the web browser can automatically keep the correct ratio of width to height. So for example, if you want to scale the image by 50%, you can simply set the width attribute and the web browser will automatically calculate the height. `` But what happens if the photo doesn't load? Perhaps the file was accidentally deleted, or you mistyped the file name. Luckily, the web browser has a way to display some text when the image fails to load. This is done using the alt attribute. For example, you can display the text My Profile Photo using the alt attribute in the previous HTML. `` It is important to ensure that screen reader accessibility software can interpret images displayed in the web browser. To support this, the `` tag is combined with the `<figure>` and `<figcaption>` tags to provide a description of the image. The `` tag is added inside the `<figure>` tag and the `<figcaption>` is specified after it.

1

2

3

4

```
<figure>



<figcaption>A photo of myself on a beach in 2015</figcaption>

</figure>
```

One last thing to note is that like videos and audio, the web browser only supports specific file types. These file types are:

- .APNG – Animated Portable Network Graphics
- .AVIF – AV1 Image Format
- .GIF – Graphics Interchange Format
- .JPEG / .JPG – Joint Photographic Expert Group image format
- .PNG – Portable Network Graphics
- .SVG – Scalable Vector Graphics
- .WEBP – Web Picture Format

Images will be important as you build websites and ensuring they are accessible will provide a better user experience for all visitors.

Completed

iFrame sandbox cheat sheet

The `<iframe>` is the inline frame element that embeds an HTML page into another page. Apart from the global attributes, which can be a part of the `iframe`, major element-specific attributes are listed below.

allow

It specifies what features or permissions are available to the frame, for instance, access to the microphone, camera, other APIs and so on. For example:

- `allow="fullscreen"` the fullscreen mode can be activated
- `allow="geolocation"` lets you access the user's location

To specify more than one feature, a semicolon-separator should be used between features. For example, the following would allow using the camera and the microphone:

```
<iframe src="https://example.com/..." allow="camera; microphone"> </iframe>
```

name

The name for the `<iframe>`. For example: `<iframe name = "My Frame" width="400" height="300"></iframe>`

height

It specifies the height of the frame. The default value is 150, in terms of CSS pixels. width

width

Specifies the width of the frame, in terms of CSS pixels. The default value is 300 pixels.

referrerpolicy

A referrer is the HTTP header that lets the page know who is loading it. This attribute indicates which referrer information to send when loading the frame resource. The common values are:

- `no-referrer` The referrer header will not be sent.
- `origin` The referrer will be limited to the origin of the referring page

- **strict-origin** The origin of the document is sent as the referrer only when using the same protocol security level (HTTPS to HTTPS)

sandbox

To enforce greater security, a sandbox applies extra restrictions to the content in the `<iframe>`. To lift particular restrictions, an attribute value (permission token) is used. The common permission tokens are listed below:

- **allow-downloads** Allows the user to download an item
- **allow-forms** Allows the user to submit forms
- **allow-modals** The resource can open modal windows
- **allow-orientation-lock** Lets the resource lock the screen orientation
- **allow-popups** Allows popups to open
- **allow-presentation** Allows a presentation session to start
- **allow-scripts** Lets the resource run scripts without creating popup windows

Note that when the value of this attribute is empty, all restrictions are applied. To apply more than one permission, use a space-separated list. For example, the following would allow form submission and scripts while keeping other restrictions active:

1

2

```
<iframe src="my_iframe_sandbox.html" sandbox="allow-forms allow-scripts">

</iframe>
```

src

The URL of the page to embed in the `<iframe>`. Using the value `about:blank` would embed an empty page.

srcdoc

Let's you specify the inline HTML to embed in the `<iframe>`. When defined, this attribute would override the `src` attribute. For instance, the following code will display "My inline html" in the frame, instead of loading `my_iframe_src.html`.

1

2

```
<iframe src="my_iframe_src.html" srcdoc="<p>My inline html</p>" >

</iframe>
```

loading

This attribute let's you specify if the iframe should be loaded immediately when the web page loads (**eager**) or loaded when iframe is displayed in the browser (**lazy**). This allows you to defer loading iframe content if it is further down your web page and outside of the display area when the page is initially loaded.

1

2

```
<iframe src="my\_iframe\_src.html" loading="lazy" >  
</iframe>
```

title

This attribute let's you add a description to the iframe for accessibility purposes. The value of this attribute should accurately describe the iframe's content.

1

2

```
<iframe src="history.html" title="An embedded document about the history  
of my family" >  
</iframe>
```

Completed

Additional resources

The following resources will be helpful as additional references in dealing with different concepts related to the topics you have covered in this module.

<https://html.com/media/>
<https://studio.support.brightcove.com/publish/choosing-correct-embed-code.html>
https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Images_in_HTML_L
<https://www.educba.com/iframes-in-html/>
<https://www.tutorialrepublic.com/html-tutorial/html5-canvas.php>

<https://html.spec.whatwg.org/multipage/media.html>

Completed

Week2-

Understanding flexbox

Much like the div and box container that you can create using HTML, flexbox is a type of container. Flexbox can overcome the limitations caused by containers such as block and inline because it does a better job of scaling over larger web pages and also provides more dynamic control of the containers. This is because it can grow, shrink and align the items inside it which give better control to the programmer over the contents and styling of the items inside the container.

Before learning about the common layouts built using the flexbox, it is important to understand the properties inside it and how flexbox works. Let's examine some of the important characteristics of flexboxes and the properties that can be used to configure them.

Flexbox is single-dimensional, which means you can align it either along a row or a column and it is set to row alignment by default. There are two axes, the main and cross-axis, much like the x and y-axis used in coordinate geometry. When aligned along the row, the horizontal axis is called the main axis and the vertical axis is called the cross axis. For the items present inside the flexbox container, the placement starts from the top-left corner moving along the main or horizontal axis. When the row is filled, the items continue to the next row. Note that with the help of a property called flex-direction, you can instead flip the main axis to run vertically and the cross axis will then be horizontal. In such a case, the items will start from top left and move down along the vertical main axis. The properties you choose will help better control alignment, spacing, direction and eventually styling of the container and items present inside it.



Image source: <https://css-tricks.com/snippets/css/a-guide-to-flexbox/#aa-basics-and-terminology>

Let's now examine some of the important properties that will allow you to configure a flexbox.

Flexbox properties

Original HTML code:

1

2

3

4

5

6

7

8

9

10

11

<body>

```
<div class="flex-container">

<div class="box box1"> One..</div>

<div class="box box2"> Two..</div>

<div class="box box3"> Three..</div>

<div class="box box4"> Four..</div>

<div class="box box5"> Five..</div>

<div class="box box6"> Six..</div>

<div class="box box7"> Seven..</div>

</div>

</body>
```

Original CSS file:

1

2

3

4

5

6

7

```
.box{  
background-color: aquamarine;  
border-radius: 5px;  
margin: 2px;  
padding: 10px;  
}
```

Output:

One..

Two..

Three..

Four..

Five..

Six..

Seven..

There are seven div containers inside the HTML file.

The corresponding CSS file contains rules for all seven div tags that have the box class. Note how two class names are given for each of the tags, one that is common among all classes and another independent of it. The style is applied to all the containers.

Now let's add properties to the flex container by converting it into flex.

display: flex;

1

2

3

4

```
.flex-container{  
    display: flex;  
}
```

The output is now seven flex containers that run from left to right starting in the top left corner.

One..

Two..

Three..

Four..

Five..

Six..

Seven..

Alignment properties

Let's examine a few alignment properties inside the flex. There are four main properties used to align a flex container and items present inside it:

- justify-content. For item alignment on main axis.
- align-items. For item alignment on cross axis.
- align-self. For unique flex items on cross axis.
- align-content. Used for packing flex lines and control over space.

Of these, justify-content and align-items are frequently used for the respective two axes.

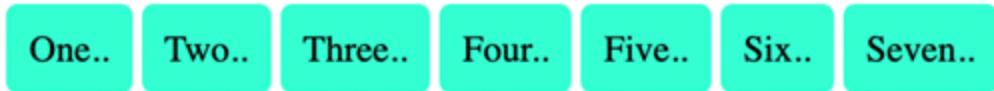
Let's first examine the use of justify-content which has a value of 'left' by default.

justify-content

CSS:

```
1  
2  
3  
4  
5  
  
.flex-container{  
  
    display: flex;  
  
    justify-content: center  
  
}
```

Output:



flex-wrap:

The default for this property is ‘nowrap’ which means the items will span the entire width of the axis.

1

2

3

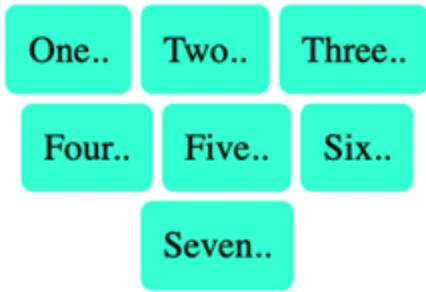
4

5

6

```
.flex-container{  
    display: flex;  
    justify-content: center;  
    flex-wrap: wrap;  
}
```

Output:



The items will now be wrapped to the size of the available viewport.

flex-direction:

This property is used to set the main axis, which is a 'row' by default. It basically means you are changing your 'main' axis from horizontal rows to vertical columns.

CSS Code:

```
1
2
3
4
5
6
7

.flex-container{
    display: flex;
    justify-content: center;
    flex-direction: column;
    flex-wrap: wrap;
}
```

Output:

One..

Two..

Three..

Four..

Five..

Six..

Seven..

The output looks like the original output; however, it is now actually a flex.

Now let's align the items again and examine a couple of the other properties mentioned earlier.

align-items:

The alignment on the cross-axis is done with the help of this property. Let's change the value for it to 'flex-end'.

CSS Code:

1

2

3

4

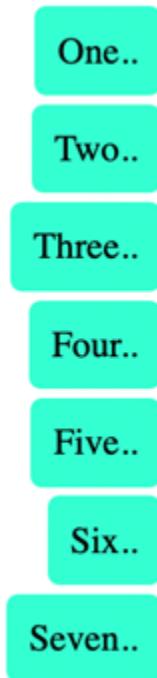
5

6

7

```
.flex-container{  
    display: flex;  
    justify-content: center;  
    flex-direction: column;  
    flex-wrap: wrap;  
    align-items: flex-end;  
}
```

Output:



The term 'end' refers to the right side of the page as the left side is seen as the beginning.

align-self:

This property can be used on individual items inside the flex.

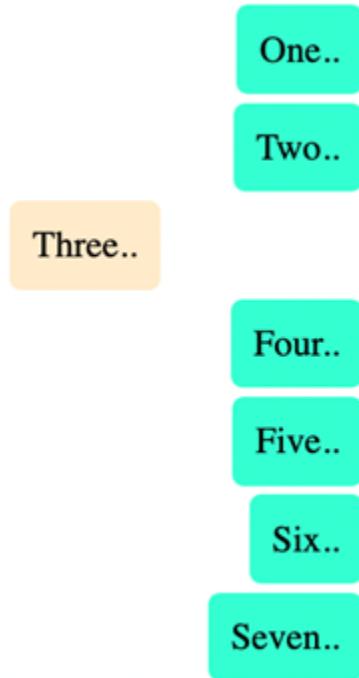
```
2
3
4
5
6
7
8
9
10
11
12

.flex-container{
    display: flex;
    justify-content: center;
    flex-direction: column;
    flex-wrap: wrap;
    align-items:flex-end;
}

.box3{
    background-color: blanchedalmond;
    align-self: center;
}
```

```
}
```

Output:



Here the color and alignment of the third box have been changed and it overrides the properties set using align-items.

gap:

gap property can be used to create space between the items along the main axis. You can also individually configure the gaps in rows and columns using row-gap and column-gap properties.

CSS Code:

1

2

3

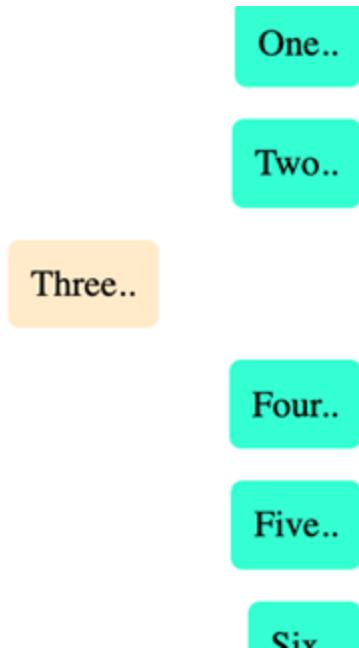
4

5

6

```
7  
8  
9  
10  
11  
12  
13  
  
.flex-container{  
    display: flex;  
    justify-content: center;  
    flex-direction: column;  
    flex-wrap: wrap;  
    align-items:flex-end;  
    gap:10px;  
}  
  
.box3{  
    background-color: blanchedalmond;  
    align-self: center;  
}
```

Output:



There is a clear change in spacing between the items.

The final set of properties are flex-grow, flex-shrink and flex-basis. Together these determine how the flex takes up space, grows or shrinks according to the space available.

These are the sub-properties of a property called flex. Together all three properties can also be given values with the help of something called the shorthand notation in CSS. Shorthand notation helps you make your code compact and also easy to write and follow. The values left empty in shorthand notation are given their default values.

For example:

```

1
2
3
4

```

```
.flex-container{
  flex: 0 1 auto;
}
```

Here for the flex-container class, there is a set rule for the flex property. The values correspond to the three properties, namely the flex-grow set to 0, flex-shrink to 1 and flex-basis to auto. The

`flex-basis` sets the initial size of the container, and together they define the rigidity or flexibility and dynamism you want to add to the flexbox.

To demonstrate the effect of this, the code has to be modified slightly by removing the flex-direction value set to 'column'. This will change it to default 'row' and the output will again be centrally aligned and horizontal best-distributed between two rows.

Output:



The rest of the remaining code is unchanged. However, the output will change if the code is modified with the addition of the `flex` property inside the `flex item box3` class.

CSS Code:

```
.box3{  
background-color: blanchedalmond;  
  
align-self: center;  
  
flex: 1 1 auto;  
}
```

Output:

One..

Two..

Three..

Four..

Five..

Six..

Seven..

The third box now takes up the entire free space available because flex-grow's value has been set to 1. So if we have `flex-grow` set to 1, the children will all set to equal size. And if one of the children has a value of 1.5, that child would take up more space as compared to the others. It's important to understand how the changes in the main and cross axis affect the way you imagine and manipulate the flexbox. Once you've had more practice you'll be more comfortable with the use of these properties, and it will become easier to use flexboxes and understand the flow and alignment of items inside the flexbox.

Completed

CSS units of measurement

A web page, as you know it, is two-dimensional. In other words, it has width and height. There are a number of other ways you can express this such as vertical and horizontal, length and breadth, x and y axis and so on. Another property of a web page is its size which can either be static or dynamic. When you've encountered enough CSS code, you will note a number of different ways in which the values for the same property can be declared using different units of measurement. Most of these units of measurement are used to account for the dynamism and dimensionality of a web page.

Let's examine the most widely used units of measurement. They can broadly be categorized as Absolute and Relative units.

Absolute units

Absolute units are constant across different devices and have a fixed size. They are useful for activities like printing a page. They are not so suitable when it comes to the wide variety of devices in use today that have different viewport sizes. Because of this, absolute units are used when the size of the web page is known and will remain constant.

The table for absolute units can be seen below:

Unit	Name	Comparison
Q	Quarter-millimeters	1Q = 1/40th of 1cm

mm	Millimeters	$1\text{mm} = 1/10\text{th of } 1\text{cm}$
cm	Centimeters	$1\text{cm} = 37.8\text{px} = 25.2/64\text{in}$
in	Inches	$1\text{in} = 2.54\text{cm} = 96\text{px}$
pc	Picas	$1\text{pc} = 1/6\text{th of } 1\text{in}$
pt	Points	$1\text{pt} = 1/72\text{nd of } 1\text{in}$
px	Pixels	$1\text{px} = 1/96\text{th of } 1\text{in}$

Of these, the pixels and centimeters are most frequently used for defining properties.

Relative values

When you create a web page, you will almost never have only a single element present inside it. Even in case of containers such as flexboxes and grids, there's usually more than one element present that rules are applied to. Relative values are defined 'in relation' to the other elements present inside the parent element. Additionally, they are defined 'in relation' to the viewport or the size of the visible web page. Given the dynamic nature of web pages today and the variable size of devices in use, relative units are the go-to option in many cases. Below is a list of some of the important relative units.

Unit	Description and relativity
<code>em</code>	Font size of the parent where present.
<code>ex</code>	x-co-ordinate or height of the font element.
<code>ch</code>	Width of the font character.
<code>rem</code>	Font size of the root element.
<code>lh</code>	Value computed for line height of parent element.
<code>r1h</code>	Value computed for line height of root element which is <code><html></code> .
<code>vw</code>	1% of the viewport width.
<code>vh</code>	1% of the viewport height.
<code>vmin</code>	1% of the smaller dimension of viewport.
<code>vmax</code>	1% of the larger dimension of viewport.

% Denotes a percentage value in relation to its parent element.

Many of these units are used in terms of the relative size of fonts. Some units are more suitable depending on the relative context. Like when the dimensions of the viewport are important, it's more appropriate to use `vw` and `vh`. In a broader context, the relative units you will see most frequently used are percentage, `em`, `vh`, `vw` and `rem`.

Much like the absolute and relative units discussed above, certain properties have their own set of acceptable values that need to be taken into account. For example, color-based properties such as `backgroundcolor` will have values such as hexadecimal, `rgb()`, `rgba()`, `hsl()`, `hsla()` and so on. Each property should be explored on an individual basis and practicing with the code will help you to decide which of these units of measurement are the most suitable choice.

Completed

Grids and flexbox cheat sheet

Note: ‘|’ stands for alternatives or OR.

Grid

The syntax for creating a grid:

1

2

3

```
selector{  
  display: grid; /* or inline-grid */  
}
```

Grid shorthand consists of the following properties with default values:

`grid`

A grid will allow you organize the various elements on your page.

`grid-template-rows: none`

This feature allows you configure your elements so that they are organized similarly to rows on a table.

`grid-template-columns: none`

This feature allows you configure your elements but with this setting the elements are organized like columns on a table.

`grid-template-areas: none`

This feature allows you configure the names of a grid and how they sit in relation to one another.

`grid-auto-rows: auto`

Default setting for all row sizes that have not been explicitly configured.

`grid-auto-columns: auto`

Default setting for all column sizes that have not been explicitly configured.

`grid-auto-flow: row`

Default location for rows that are not explicitly allocated.

`column-gap: normal`

This sets the gap between the columns

`row-gap: normal`

This sets the gap between the rows

Grid properties for container

`grid-template-columns: measurement units | % units | repeat()`

Defines the line names, and maintains a constant size of column items. Can accept a range of different measurement sizes.

`grid-template-rows: measurement units | % units | repeat()`

Defines the line names, and maintains a constant size of rows. Can accept a range of different measurement sizes.

`grid-auto-columns: measurement unit (fixed value for all columns)`

Determines the default size for columns that have not been explicitly configured.

`grid-auto-rows: measurement unit (fixed value for all rows)`

Determines the default size for rows that have not been explicitly configured.

`grid-template: "header header" auto`

This allows you define and maintain named cells on a grid

`"main right" 75vh`

This defines two cells named main and right, that have a sizing of 75% of the viewport height.

`"footer footer" 20rem`

This defines two cells named footer and footer, that have a sizing of 20 root em (rem). This defines the size in relation to the html font size.

Gap

`grid-gap: measurement units`

Determines the gap between rows and columns

`grid-column-gap: measurement units`

Determines the gap between columns

`grid-row-gap: m-unit-1 m-unit-2`

Determines the gap between columns

Alignment

`justify-items: start | center | end | stretch`

Defines the default space that is allot to each item on the grid

`align-items: start | center | end | stretch`

Defines the default space related to an item along the grid's block axis

`place-items: start | stretch /* shorthand for two properties above */`

This feature allows you align items with the block and inline directions.

Justification

```
justify-content: start | center | end | stretch | space-between | space-evenly  
| space-around
```

Defines browser allocation of space to content items in relation to the main-axis

```
align-content: start | center | end | stretch | space-between | space-evenly |  
space-around
```

Defines browser allocation of space to content items in relation to cross axis and block axis

```
place-content: center | start
```

This feature allows you align items with the block and inline directions.

Positioning

```
grid-auto-flow: row | column | dense
```

This relates to how the items are placed automatically within the grid

```
grid-auto-columns: measurement units
```

This relates to the size for columns created without specific size specifications

```
grid-auto-rows: measurement units
```

This relates to the size for rows created without specific size specifications

Grid properties for items (child)

```
grid-column: column position /* E.g. 1/2 */
```

Allows for specifying where on the grid the column is to start.

```
grid-column-start: column start position
```

This property determines the starting column position an item is placed on a grid.

```
grid-column-end: column end position
```

This property determines the end column position an item is placed on a grid.

```
grid-row: row position /* E.g. 1/2 */
```

Allows for specifying where on the grid the row is to start.

```
grid-row-start: row start position
```

This property determines the starting row position an item is placed on a grid.

```
grid-row-end: row end position
```

This property determines the end row position an item is placed on a grid.

Justification and alignment

```
justify-self: start | center | end | stretch
```

Determines how an item is positioned inside its aligned container in relation to the appropriate axis.

```
align-self: start | center | end | stretch
```

Aligns an item within a grid area.

```
place-self: start | stretch /* shorthand for two properties above */
```

This setting lets one align and justify an item within a block.

Flexbox

The syntax for creating a flexbox:

1

2

3

4

```
selector{  
    display: flex | inline-flex  
}
```

Here the selector can refer to any of the following flex attributes

- Attribute selector
- Class Selector
- ID Selector
- Type Selectors
- Universal Selectors

The display relates to how you want the selector to be shown. Setting display to flex makes the given selector a flex box. Setting display to `inline-flex` makes the selector a flex box container while will be inline.

Properties for flexbox container

`flex-direction: row | row-reverse | column | column-reverse`

It is possible to specify the direction your elements will follow. Traditionally text goes from left to right which is flex's default setting however it can be set from right to left or even top to bottom. The four flex-direction are:

- row : organized from left to right
- row-reverse: organized from right to left
- column: organized from top to bottom
- column-reverse: organized from bottom to top.

`flex-wrap: wrap | nowrap`

The standard layout is to plot the elements from left to right in a straight line. The wrap feature allows you customize this to match the size of the window displaying the page.

- wrap: Automatically wrap the items with as the window space gets smaller.

- Nowrap: Default setting, items remain rigid and don't respond to adjustments made to the window size.

align-items: flex-start | flex-end | center | stretch

This determines how the flex items are to be positioned on the page. Items can be aligned in a variety of ways

- Flex-start: Similar to standard writing, items start at the top left-hand corner and are positioned from left to right
- Flex-end: Position begins in the bottom right hand corner.
- Center: Item is positioned from the center.
- Stretch: item expands to fill the container.

justify-content: flex-start | flex-end | center | space-between | space-around

Justify-content determines the alignment of the flex items.

- Flex-start: goes from right to left along the main axis.
- Flex-end: goes from left to right along the main axis.
- Center: Starting at the middle, alignments expands from there.
- Space-between: first and last item are flush with the left and right wall respectively, every other item is evenly spaced.
- Space-around: each item is equidistant from each other and the boundary wall

Properties for flexbox items (child)

flex-grow: factor of flex's main size

This attribute enables the flex container to grow proportionally to the other containers present.

flex-shrink: factor of flex's main size

This allows elements to shrink in relation to items around it.

flex-basis: auto | factor of main's size | measurement unit

The sets the initial main size of an item. It can be overridden if other stylized elements are configured.

order: position in flex /* Set ascending by default */

The standard positioning of items is by source order, however this feature will enable you to configure where the items appear on the page.

align-self: start | center | end | stretch

This determines where on the page the child items will be positioned. Similar to the main flex attributes, start is to the left and end is to the right.

Completed

Grid template area

Grid areas are a way to group one or more grid cells. The grid template area is an extension of this concept where you can give names to these grid areas. Once you have the names defined, you can address these new grid area items by their names and configure them accordingly.

The property grid-template-areas is usually placed inside the body tag or any container where the grid needs to be placed, the same way that you would define the rules for the grid. The main difference is, in case of grid-template-areas the values present will be the different names.

Process

The process isn't prescriptive but these are the steps in general:

- Define the grid using display property
- Set the height and width of the grid
- Set the grid-template-areas with the appropriate name identifiers
- Add the appropriate sizes for the rows inside the grid using grid-template-rows property
- Add the appropriate sizes for the columns inside the grid using grid-template-columns property

But how exactly do you use these names and where do they come from? The names that you use inside the grid template areas are the HTML tags that you have used. Or, where you need to get more specific, you designate a class name to these tags. Once the names are assigned, you define the properties for each class the same way that you define them conventionally. Let's examine an example.

Example

HTML Code:

1

2

3

4

5

6

7

8

9

10

```
1 <div></div>
2 <div></div>
3 <div></div>
4 <div></div>
5 <div></div>
6 <div></div>
7 <div></div>
8 <div></div>
9 <div></div>
10 <div></div>
```

11

12

13

14

15

16

17

18

19

<head>

<link rel="stylesheet" href="gridta.css">

</head>

<body>

<header> Header </header>

```
<nav class="nav-bar"> Navigation </nav>

<main> Main area </main>

<footer> Footer </footer>

</body>
```

CSS Code:

1
2
3
4
5
6
7
8
9
10
11
12
13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
body {  
  
    display: grid;  
  
    height: 200px;  
  
    grid-template-areas: "head head"  
                        "nav   main"  
                        "footer footer";  
  
    grid-template-rows: 30px 1fr 30px;  
  
    grid-template-columns: 150px 1fr;
```

```
}
```

```
header {
```

```
    grid-area: head;
```

```
    background-color: lightsalmon;
```

```
}
```

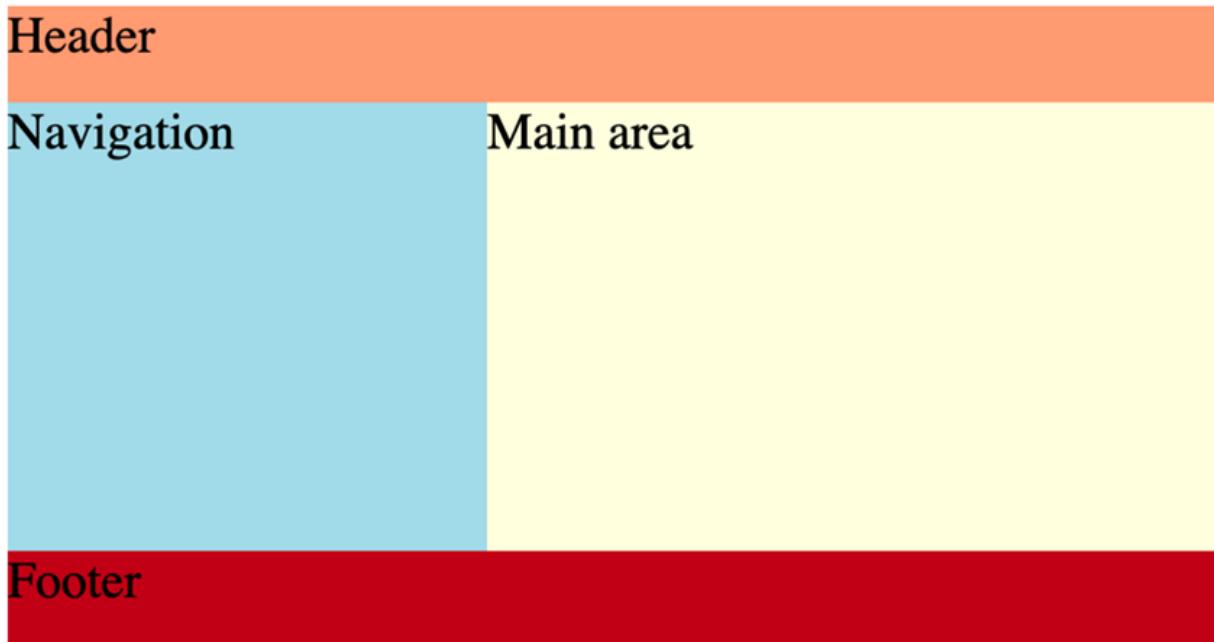
```
.nav-bar {
```

```
    grid-area: nav;
```

```
    background-color: lightblue;
```

```
}
```

Output:



Though there are five sets of rules, logically the CSS code is divided into two sections. The first is where you define the rules for the grid inside the body selector. And second is where you allocate specific rules for the different grid areas. The way these grid areas are distributed is according to how you have defined the names inside the grid-template-areas property. In the example above the relevant code is:

1

2

3

```
grid-template-areas: "head head"  
                      "nav  main"  
                      "footer  footer";
```

The ‘head’ is written twice to imply two columns and the rest of the content follows the usual convention. The number of rows will be the number of ‘pairs of quotes’ you have used which in this case is three. Namely “head head”, “nav main” and “footer footer”. Once you know the number of rows and columns, the values for those will be set by grid-template-rows and grid-template-columns. Since these are three and two respectively here, you need to add that many values. The height simply gives the overall value of the height for the grid.

Note that the number of times you wrote “header” did not have to be two. You could write more of those and if you align the rest of the grid-names correctly, the height and width of the grid-areas will be distributed proportionately.

Let’s return to the example. If you keep all other properties the same but you change the grid-template-areas as follows:

1

2

3

```
grid-template-areas: "head head head"  
                      "nav   main main"  
                      "footer footer footer";
```

The output will remain the same as you have fixed the value of the third row to “30px”. The example is simple for the sake of clarity, but if you had used relative values, you would’ve seen an observable change in the comparable sizes of nav and main grid-areas.

Grid-areas are convenient when you have a clear schematic of what you want in a grid. It’s also easier to configure individual areas if you can address them by their names. Let’s say you are designing a resume on your website, you will be able to name the different areas such as ‘Bio’, ‘Education’, ‘Work experience’ and so on. And it’s easier to use these labels when you are defining the rules. Creating a block diagram using pen and paper before starting to work on a grid is always a good idea.

Completed

Create a grid layout (solution)

Below is a sample CSS solution file for the previous exercise.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

```
.container {  
    display: grid;  
    max-width: 900px;
```

```
min-height: 50vh;

grid-template-columns: 100%;

grid-template-rows: auto auto 1fr auto auto;

grid-template-areas: "header" "left" "main" "right" "footer";

}

@media (min-width: 440px) {

.container {

grid-template-columns: 150px 1fr 150px;

grid-template-rows: auto 1fr auto;

grid-template-areas: "header header header" "left main right" "footer footer";

}

}

.header {

grid-area: header;

padding: 10px;

background-color: black;

color: #fff;

text-align: center;
```

```

}

.main {
  grid-area: main;
  padding: 25px;
}

.left {
  grid-area: left;
  background-color: peachpuff;
}

.right {
  grid-area: right;
}

.footer {
}

```

While reviewing the code, note the following:

- The grid template areas are defined as "header" "left" "main" "right" "footer" but for a small device with a screen width of 440px or less, it is defined as "header header header" "left main right" "footer footer footer" using a media query.
- The `grid-rows` property value also changes based on the media query.
- The values for the number of rows you add for `grid-template-rows` and number of columns you add for `grid-template-columns` must match the dimensions of the grid-template-areas.

- `grid-area` that has undefined rules will appear empty. (Does not happen with the example above.)
- Each CSS rule specifies which grid area they belong to by using the `grid-area` CSS property.
- The selectors of each rule used are element tags in HTML or classes, as we have used here.

Completed

Additional resources

Here is a list of resources about layouts, flexboxes, grids and viewports in HTML and CSS that may be helpful as you continue your learning journey.

[Broad overview of layouts in CSS](#)

[Detailed overview of flexboxes](#)

[Detailed overview of grids \(1\)](#)

[Detailed overview of grids \(2\)](#)

[Viewports in CSS](#)

[Grid layout applications](#)

[Overview of different layouts](#)

Completed

All selectors and their specificity

As you build a website, the complexity of the code might increase to such a point that more than one CSS rule is applied to the same element. Or, you might accidentally add more than one rule over the same element. This results in conflicts as only one rule can be applied to a specific property. For example, the color of a certain p tag can either be blue or white, but not both. CSS engines use something called specificity to resolve these conflicts. Specificity is the ranking or score that helps CSS determine the final rule that will be applied to a given element.

This reading will help you grasp how the element with the ‘highest’ specificity is selected by CSS. But before you read on, it is important to note that these rules only apply in cases where conflicts arise for the properties.

Specificity hierarchy

CSS has a set of rules that it uses to ‘score’ or assign a certain weight to selectors and this creates a specificity hierarchy. Based on the weights, there are four categories in this hierarchy:

- Inline styles
- IDs
- Classes, attributes, and pseudo-classes
- Elements and pseudo-elements

Inline styles

Inline styles are attached to the elements within your HTML code like with the 'style' attribute. inline styles have the highest specificity. That means that rules that are defined as inline styles will be applied irrespective of other rules.

For example, take these two rules that create a conflict in color styling for a p tag:

1

2

```
<p style="color: white;">  
  
p{color: blue}
```

The p tag will be colored white because it is declared inside the inline tag.

IDs

Next in the hierarchy are IDs and by now you know that they are represented by '#'. For example:
#div

Classes, attributes, and pseudo-classes

Classes, and the attributes inside the selectors, come next with what is called the pseudo-classes that you will soon learn more about.

For example:

```
.my-class  
p["attribute"]  
div:hover
```

Elements and pseudo-elements

Finally, elements and something you call pseudo-elements have the lowest position in the specificity hierarchy. You will learn more about pseudo-elements later in this lesson.

Calculating scores

But by now you might wonder how is specificity calculated?

CSS uses the hierarchical model internally to calculate the specificity of the selectors used on a web page. But often as the size of CSS code increases, developers unavoidably face rule conflicts. In these cases, developers use the specificity hierarchy to calculate the precedence of CSS rules and to control the outcome of their web pages.

Let's explore a practical example of how to determine the score of a few selectors.

```
#hello {} will be 0100  
div {} will be 0001 and  
div p.foo {} will be 0012
```

In the order stated above, the four categories will be assigned values 1000, 100, 10 and 1 with the element selectors having the lowest value of 1. These scores will be calculated respectively for each element present inside the selector. The total score for these elements is then calculated and the one with the highest value has the highest specificity.

Let's explore a couple of examples for clarity. Take note that the properties and values are not included in these examples to keep the focus on the selectors only.

Example 1

1

2

3

```
p {}  
  
div p {}  
  
div p.foo {}
```

p => 1 element => 0 0 0 1 => Score: 1
div p => 2 elements => 0 0 0 2 => Score: 2
div p.foo {} => 2 elements and 1 class selector => 0 0 1 2 => Score: 12

The third case has a total of 12 for the p tag and so has the highest specificity. The rules for the other two cases are then overridden and the rules inside the third case are applied.

Example 2

```
p#bar => 1 element & 1 ID => 0 1 0 1 => Score: 101  
p.foo => 1 element & 1 class => 0 0 1 1 => Score: 11  
p.p.foo => 1 element & 2 class => 0 0 2 1 => Score: 21
```

By now it should be clear that the case containing ID has a much higher score and the rules inside it will be applied.

Once you learn about the different pseudo-classes, pseudo-elements, and wide range of selectors later in this section, it will be easy to see why understanding specificity is important.

While the weights assigned from the hierarchical structure help in a systematic approach, there are a few more guidelines and rules that become important especially in cases when the score for the different selectors is the same. Some of these are:

- Every selector will have a score and place in the hierarchy
- In the case of selectors with equal specificity, the latest or last written rule is the one that will be applied
- In general, ID selector should be applied in cases where you need to be certain about a rule

- Universal selectors have zero specificity value

This reading only gave you an overview of specificity, but you should know that it is a much broader topic and also the underlying basis on which CSS engines work. That's what the 'Cascading' in CSS means: the way in which CSS engines evaluate and apply the specificity rules is called 'cascade'. Cascade is a type of small waterfall that falls in stages down the rocks and that is exactly how CSS behaves.

Don't be too worried about applying what you've learned now, there are CSS specificity calculators available that can help you with determining the styling outcomes of your pages.

Completed

Targeted CSS (solution)

Below is a sample CSS solution file for the previous exercise.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

* {

font-family: Monaco;

}

.menu, .ll {

text-align: center;

color: #fa9f42;

}

.ll {

font-size: 30px;

margin-bottom: 20px;

border-bottom: 2px solid #495e57;

}

```
.menu-container {  
    max-width: 800px;  
    display: flex;  
    justify-content: center;  
    background-color: #e0e0e2;  
}  
  
.section {  
    padding: 10px;  
}  
  
.label {  
    font-weight: bold;  
}  
  
.description {  
    font-style: italic;  
    border-top: 2px solid #495e57;  
}
```

```
.item-name {  
  margin: 25px;  
  font-size: 12px;  
}
```

While reviewing the code, note the following:

- For div elements that follow the h3 element, the child combinator is defined as `div > h3`
- For elements using the CSS class `low` that follow the `label` CSS class, the adjacent sibling combinator is defined as `.label + .low`
- For div elements that follow other div elements, the general sibling combinator is defined as `div ~ div`

Completed

Pseudo-elements

By now you know that pseudo-elements help you style a specific part of an element. For example, you can decide to apply styling to only the first word or line in a given element. First, let's examine the syntax of a pseudo-element.

Syntax

```
selector::pseudo-element {  
  property: value;  
}
```

It is important to note that pseudo-elements use two colon characters instead of one.

Now, let's explore some examples of popular pseudo-elements.

::first-letter

You can use `first-letter` to change the color of just the first letter of each of the three points in the example text.

HTML code:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

```
<!DOCTYPE html>

<html>

<head>
    <link rel="stylesheet" href="pseudo4.css">
</head>

<body>
    <ul>
        <li>Based in Chicago, Illinois, Little Lemon is a family-owned Mediterranean restaurant, focused on traditional recipes served with a modern twist. </li>
        <li>The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12-15 items that they rotate seasonally. The
    </ul>
</body>

```

restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy.

</body>

</html>

CSS code:

1

2

3

4

5

6

7

8

9

10

11

```
li::first-letter {
```

```
color:coral;
```

```
font-size: 1.3em;
```

```
font-weight: bold;
```

```
line-height: 1;
```

```
}
```

Output

- Based in Chicago, Illinois, Little Lemon is a family owned Mediterranean restaurant, focused on traditional recipes served with a modern twist.
- The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.
- Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy.

Although the code only changed the first letter of each bullet point, it makes a big difference in terms of presentation. Now let's change the font in a different way.

::first-line

First-line will change the complete first line of each of the bullet points to light sea green.

CSS code:

1

2

3

4

5

6

7

8

9

10

```
ul {  
    list-style-type: none;  
}  
  
li::first-line {  
    color: lightseagreen;  
    text-decoration: underline;  
    line-height: 1;  
}
```

Output:

Based in Chicago, Illinois, Little Lemon is a family owned Mediterranean restaurant, focused on traditional recipes served with a modern twist.

The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy.

Because it's only the first line of each bullet point, it almost functions like dividers between the three different points instead of having to rely on bullets.

Note that the contents of the line to which this pseudo-element is applied will change as you increase or decrease the size of your viewport.

Output:

Based in Chicago, Illinois, Little Lemon is a family owned Mediterranean restaurant, focused on traditional recipes served with a modern twist.

The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy.

::selection

Selection is another useful pseudo-element. For example, you may use it when you are taking notes on your device because it allows you to highlight specific text. The effect of it becomes obvious only after the user selects content. On web pages today, you will typically see inverted colors from white-black to black-white when selecting a portion of text.

CSS code:

1

2

3

4

5

6

7

8

9

10

```
ul {  
    list-style-type: none;  
}  
  
li::selection {  
    color: brown;  
    background-color: antiquewhite;  
    line-height: 1;  
}
```

Here is an example of a selection of text.
Output:

Based in Chicago, Illinois, Little Lemon is a family owned Mediterranean restaurant, focused on traditional recipes served with a modern twist.

The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy.

And another example of the same text but with a different section selected and highlighted.

Output:

Based in Chicago, Illinois, Little Lemon is a family owned Mediterranean restaurant, focused on traditional recipes served with a modern twist.

The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy.

Different segments of the text are highlighted depending on the text that is selected at any given point.

::marker

Markers are typically used to add style elements to a list, for instance, to color bullet points. For example, you can enhance the user experience when you use a marker in the following way.
CSS code:

```
1
2
3
4
5
6

li::marker {
    color: cornflowerblue;
    content: '<> ';
    font-size: 1.1em;
}
```

Output

- ◊ Based in Chicago, Illinois, Little Lemon is a family owned Mediterranean restaurant, focused on traditional recipes served with a modern twist.
- ◊ The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Now the bullet points are cornflower blue and they have the shape specified in the code.

::before and ::after

One more pair of pseudo-elements are the **::before** and **::after** pseudo-elements. They allow you to add content before and after an element on which they are allowed. In other words, new content can be added to a page without adding HTML code for it. You can also add styling options for this content. Let's do an example where text is added both before and after some cooking guidelines to identify them as important tips.

HTML code:

```
1<body>  
2  <p id="tips"> Don't rinse your pasta after it is drained. </p>  
3  
4  
5  
6
```

<body>

```
<p id="tips"> Don't rinse your pasta after it is drained. </p>
```

```
<p> Slice the tomatoes. Take the extra efforts to seed them. </p>  
  
<p id="tips"> Peel and seed large tomatoes. </p>  
  
</body>
```

CSS code:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16
```

```
#tips::before{  
  
background: darkkhaki;  
  
color:darkslategray;  
  
content: "Tip:";  
  
padding-left: 3px;  
  
padding-right: 5px;  
  
border-radius: 10%;  
  
}  
  
 
```

```
#tips::after{  
  
background:darkkhaki;  
  
color:darkslategray;  
  
content: "!!";  
  
padding-right: 5px;  
  
border-radius: 20%;  
  
}
```

Output:

Tip: Don't rinse your pasta after it is drained. !!

Slice the tomatoes. Take the extra efforts to seed them.

Tip: Peel and seed large tomatoes. !!

The “content” property is where the text for the guidelines goes. The word “tip” has been added before each guideline thanks to the rules added for **tips::before**. And, each of the three guidelines now has two exclamation marks after them thanks to the rules added for **tips::after**. Note how the second `<p>` element inside the HTML code remains unaffected. You don’t have to use after and before together like this, but sometimes it is useful to combine them.

The examples covered here illustrate that adding simple code for pseudo-elements can greatly enhance the appearance of websites. There are plenty of other pseudo-elements and some of them are more popular than others. You can follow your own style and explore the creative possibilities that pseudo-classes and pseudo-elements offer.

Completed

CSS Pseudo cheat sheet

Simple selectors

Selector	Syntax	Example
Element	element	<code>div { } </code>
Class	.class	<code>.alpha { } </code>
ID	#id	<code>#alpha { } </code>
Universal	*	<code>* { } </code>

Variations of simple selectors

Elements	Syntax	Example	Description
Two classes	.first-class.second-class	.alpha .beta { }	All elements with classes alpha and beta
Element and class	element.class	p.alpha { }	All alpha class elements inside <p>
Two elements	element, element	p, div { }	All <p> and <div> elements
Two elements	element element	p div { }	All <div> elements inside <p>

Descendant selectors/combinators

Selector	Syntax	Example	Description
Descendant	element element	div p { }	All <p> descendants of <div>
Child	element>element	div > p { }	All <p> direct descendants of <div>
Adjacent Sibling	element+element	div + p { }	<p> element directly after <div>
General Sibling	element~element	div ~ p { }	All <p> element iterations after <div>

Attribute selectors

Selector	Syntax	Example
[attribute]	[href] { }	Selects all elements with a href attribute
[attribute=value]	[lang="fr"] { }	Selects all elements with lang attribute that has a value of "fr"
[attribute~=value]	[input~=hello] { }	Elements with input attribute containing the whitespace separated substring "hello"
[attribute =value]	[lang =en] { }	Elements with lang attribute value equal to "en" or "en-"(en hyphen)

[attribute^=value]	<code>a[href^="https"] { }</code>	Every <a> element with href attribute value begins with "https"
[attribute\$=value]	<code>a[href\$=".docx"] { }</code>	Every <a> element with href attribute value ends with ".docx"
[attribute*=value]	<code>a[href*="meta"] { }</code>	Every <a> element with href attribute value has substring "meta"

Pseudo-class	Example	Description of selection
:active	<code>a:active { }</code>	All active links
:checked	<code>input:checked { }</code>	All the checked <input> elements
:default	<code>input:default { }</code>	All default <input> elements
:disabled	<code>input:disabled { }</code>	All disabled <input> elements
:empty	<code>div:empty { }</code>	All the <div> elements with no children
:enabled	<code>input:enabled { }</code>	All the enabled <input> elements
:first-child	<code>p:first-child { }</code>	All the <p> elements who are the first child of a parent element
:first-of-type	<code>p:first-of-type { }</code>	All the <p> element who are the first <p> element of a parent element
:focus	<code>input:focus { }</code>	Input element under focus
:fullscreen	<code>:fullscreen { }</code>	The element in full-screen mode
:hover	<code>p:hover { }</code>	Action effect on mouse hover
:invalid	<code>input:invalid { }</code>	Input elements with an invalid value
:last-child	<code>p:last-child { }</code>	All the <p> elements who are the last child of a parent element
:last-of-type	<code>p:last-of-type { }</code>	All the <p> elements who are the last <p> element of a parent element

:link	<code>a:link { }</code>	All unvisited links
:not(selector)	<code>:not(div) { }</code>	All the elements that are not a <div> element
:nth-child(n)	<code>div:nth-child(3) { }</code>	All the <p> elements that are the third child of a parent element
:nth-last-child(n)	<code>div:nth-last-child(3) { }</code>	All the <div> elements which are the third child of a parent element, counting from last child element
:nth-last-of-type(n)	<code>p:nth-last-of-type(2) { }</code>	The second sibling from the last child of a parent element.
:nth-of-type(n)	<code>p:nth-of-type(2) { }</code>	The second sibling of a parent element.
:only-of-type	<code>p:only-of-type { }</code>	All the <p> elements which are only <p> elements inside its parent
:only-child	<code>p:only-child { }</code>	All the <p> elements which are only child of a parent element
:optional	<code>input:optional { }</code>	The input elements with no "required" attribute
:required	<code>input:required { }</code>	Selects input elements with the "required" attribute specified
:root	<code>:root { }</code>	The Root element of document
::selection	<code>::selection { }</code>	The portion of an element that is selected by a user
:valid	<code>input:valid { }</code>	All the input elements with a valid value
:visited	<code>a:visited { }</code>	Selects all visited links

Pseudo-element selectors

Syntax	Example	Description
::after	<code>p::after { }</code>	Inserts content after content of <p> element
::before	<code>p::before { }</code>	Inserts content before content of <p> element

::first-letter	<code>p::first-letter { }</code>	Selects first letter of every <p> element
::first-line	<code>p::first-line { }</code>	Selects first line of every <p> element
::placeholder	<code>input::placeholder { }</code>	Selects input elements with "placeholder" attribute specified
::marker	<code>::marker { }</code>	Selects markers in a list

Completed

Additional resources

Here is a list of resources about selectors, pseudo-classes and pseudo-elements in HTML and CSS that may be helpful as you continue your learning journey.

[Commonly used selectors](#)

[Combinator selectors](#)

[Comprehensive list of selectors](#)

[Comprehensive list of pseudo-classes](#)

[Comprehensive list of pseudo-elements](#)

Completed

Text effects cheat sheet

The effects developers use on text items on a web page are chosen mainly because of their styling and layout style. Interesting effects can be created by combining these with other CSS properties. The visual representation of text content can be changed by four main properties: text-transform, font-style, font-weight and text-decoration.

Property	Values	Description
----------	--------	-------------

Text-transform	None, uppercase, lowercase, capitalize, full-width	Modify text properties
Font-style	Normal, italic, oblique	Font styling options such as italics
Font-weight	Normal, weight, lighter, bolder, 100-900	Other font styling options like change of emphasis such as making text bold
Text-decoration	None, underline, overline, line-through	Shorthand for auxiliary elements added to text using other properties such as text-decoration-line

The additional properties that help configure styling effects are below.

Text-align	For horizontal alignment of text
Text-align-last	Alignment for the last line when text set to justify
Text-combine-upright	Multiple characters into the space of a single character placed upright like in Mandarin
Text-decoration-color	Color configuration of the text-decoration
Text-decoration-line	Line type in text-decoration such as underline, overline and so on
Text-decoration-style	Styles added to lines under text such as wavy, dotted and so on
Text-decoration-thickness	Thickness of the decoration line
Text-emphasis	Shorthand for other properties such as color and style
Text-indent	The indentation of the first line
Text-justify	Specifies the justification method used when text-align is "justify"
Text-orientation	Orientation of text in a line such as sideways, upright and so on
Text-shadow	Adds shadow to text
Text-underline-position	Declare position of underline set using the text-decoration property

Other than these, there are some more properties that help modify the alignment and define the scope of text with their containers.

Property	Values	Description
Text-overflow	Clip, ellipsis	Determines overflow behavior of text with the container
Word-wrap	Normal, anywhere, break-word	Applies to inline elements, alias for overflow-wrap
Word-break	Normal, break-all, keep-all, break-word	Used for long words to decide if words should break or overflow
Writing-mode	Horizontal-tb, vertical-lr, vertical-rl	Can set the text direction vertical or horizontal

The properties mentioned are ones that can be used for giving effects to text.

Completed

Transforming and transitioning elements (solution)

Below is a sample CSS solution file for the previous exercise.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

* {

padding: 0;

margin: 0;

color: pink;

}

.container {

min-height: 100vh;

background-color: bisque;

```
}

.letters {
    display: flex;
    justify-content: center;
    padding-top: 20px;
    margin-bottom: 20px;
    text-transform: uppercase;
    font-size: 60px;
}

.letters p {
    text-align: center;
    background-color: rgb(250, 150, 100);
    width: 70px;
    margin-right: 1.5px;
    border-radius: 15%;
    border-color: rgb(250, 125, 75);
    border-style: solid;
}
```

```

.letters:hover p:nth-child(even) {

    transform: rotateY(360deg);

    transition: 0.5s;

}

.letters:hover p:nth-child(odd) {

    transform: rotateX(360deg);

    transition: 1.5s;

}

```

While reviewing the code, note the following:

- The transition properties specify the duration of the transform property being applied. The elements will start at zero degrees and transition to 360 degrees over the duration of the transition.
- The even and odd pseudo-classes are used in two separate CSS rules.

Completed

CSS keyframes

In this reading, you will become acquainted with the `@keyframes` rule and you'll learn how to use it with the `animation` property in CSS. The rules covered so far are for alignment and styling of web pages using CSS. Keyframes are a type of at-rule which are represented by the '@' suffix. At-rules are statements inside CSS that describe how to behave or perform certain actions. In line with that, keyframes are defined as '`@keyframes`' in the CSS code. `@keyframes` are part of the animation sequence and help in defining the steps inside it. Imagine an object on your web page moving from point A to point B. You can use the `transition` and `transform` properties to do that, but animation sequences are used to accomplish more complex behaviors in an easier way.

from{} and to{} keywords and using percentages(%) syntax

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17
```

```
@keyframes animation-name {
```

```
from {
```

```
property-a: value-a;
```

```
}
```

```
to {
```

```
property-a: value-b;
```

```
}
```

```
}
```

The 'from' and 'to' keywords are used inside the @keyframes rule to mark the transition of one or more properties and can be seen as the start and end points of that transition. As can be seen from the syntax, the values of 'property-a' changes from 'value-a' to 'value-b'. To expand on the use of 'from' and 'to', the @keyframes allows you to add more steps to your animation by using a percentage that represents the completion of the animation.

1

2

3

4

5

6

7

8

9

10

11

```
@keyframes identifier {  
    0% {transform: rotate(100deg);}  
  
    30% {opacity: 1;}  
  
    50% {opacity: 0.50;}  
  
    70% {opacity: 1;}  
  
    100% {transform: rotate(50deg);}
```

The different percentages used in the example demonstrate the progression of the animation. Note that it doesn't have to be the same property that you modify in these steps. That's not possible using the transition property, but you can do it with @keyframes. This flexibility is what makes @keyframes so powerful. Another advantage is how these animations can also loop infinitely, run forwards, reverse and alternate.

@keyframes are tied in with the animation-name to which they are going to be applied. To give an overview of the animation property, it consists of other sub-properties. Of these, animation-name and

animation-duration must be defined while others such as timing-function, delay, direction, fill-more, iteration-count and so on can be added.

Animation property shorthand:

The shorthand for the animation property consists of the following properties with their default values:

- animation-name: none
- animation-duration: 0s
- animation-timing-function: ease
- animation-delay: 0s
- animation-iteration-count: 1
- animation-direction: normal
- animation-fill-mode: none
- animation-play-state: running
- animation-timeline: auto

If the values of any of these are not defined, you should assume that they are the default. Of these, the first property of the animation-name is what's used to tie it to the @keyframes rule.

Let's now examine an example of how you can use @keyframes and the animation property.

Animation example

HTML code:

```
1 <body>  
2   <div class="box"></div>  
3 </body>  
4  
5 .box {  
6   width: 100px;  
7   height: 100px;
```

```
</body>
```

```
</html>
```

CSS code:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

body{

padding: 30px; }

.box{

```
background-color: lightcoral;  
  
width: 50px;  
  
height: 50px;  
  
animation: myanimation 3s infinite ease-in;
```

}

```
@keyframes myanimation{
```

```
from{width: 50px;
```

}

```
to{width: 100px;  
}  
}
```

And this is the output:

Play Video

In the example, the width of the object changes from 50 pixels to 100 pixels over a span of 3 seconds and loops infinitely afterward.

This is a very simple example of how you can use the animation property with the help of @keyframes rule to create your desired animation.

If you modify this code and change the animation rules to percentages, it will have the same output.

```
1  
2  
3  
4  
5  
6  
7  
  
@keyframes myanimation{  
0%{width: 50px;  
}  
  
100%{width: 100px;  
}  
}
```

Now, let's add an intermediary step at 50%.

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
  
@keyframes myanimation{  
0%{width: 50px;  
}  
  
50%{background-color: aquamarine;  
height: 20px;  
}  
  
100%{width: 100px;  
}
```

Output:

[Play Video](#)

The output is drastically different now after adding just two lines of code. Just like this, you can keep adding steps inside your @keyframes rule to make it even more dynamic and add the desired animation effects. Animation property and @keyframes rule can be used in very creative ways to enhance a web page.

Completed

Animation and effects cheat sheet

Transform property

Syntax

transform: transform function-values

Example

```
.sample-class {  
    transform: rotate(60deg);  
}
```

Keyword-value type: none

[1](#)
[2](#)
[3](#)

[1](#)
[2](#)
[3](#)

```
.sample-class {  
    transform: none;  
}
```

Function-value type: matrix()

Variations: matrix(), matrix3d()

1

2

3

```
.sample-class {  
    transform: matrix(1.0, 2.0, 3.0, 4.0, 5.0, 6.0);  
}
```

Function-value type: rotate(deg)

Variations: rotate(), rotate3d(), rotateX(), rotate(), rotateZ()

1

2

3

```
.sample-class {  
    transform: rotate3d(3,2,1, 100deg);  
}
```

Note: In rotate3d(), the respective values represent x, y, z co-ordinate and degree of rotations

Function-value type: translate(x,y)

Variations: translate(), translate3d(), translateX(), translateY(), translateZ()

1

2

3

```
.sample-class {  
    transform: translate3d(10px, 20px, 30px);  
}
```

Note: In translate3d(), the respective values represent translation along the x, y, z co-ordinates

Function-value type: scale(factor)

Variations: scale(), scale3d(), scaleX(), scaleY(), scaleZ()

[1](#)

[2](#)

[3](#)

```
.sample-class {  
    transform: scale3d(2, 1, 0.3);  
}
```

Note: In scale3d(), the respective values represent scaling times along the x, y, z co-ordinates

Function-value type: skew(deg, deg)

Variations: skew(), skewX(), skewY()

[1](#)

[2](#)

[3](#)

```
.sample-class {  
    transform: skew(100deg);  
}
```

Global value types:

[1](#)

[2](#)

[3](#)

```
.sample-class {  
    transform: inherit;  
}
```

1

2

3

```
.sample-class {  
    transform: initial;  
}
```

1

2

3

```
.sample-class {  
    transform: revert;  
}
```

1

2

3

```
.sample-class {  
    transform: revert-layer;  
}
```

1

2

3

```
.sample-class {  
    transform: unset;  
}
```

Multiple transform over the same element

Syntax

Transform can be applied for rotate(), scale() and translate() that can be listed together. Each of these properties can have their own values and the actions will give a combined effect.

Example

1

2

3

```
.sample-class {  
    transform: rotate(45deg) scale(1.5) translate(45px);  
}
```

Additional property under transform:transform-origin
Determines the anchor point for the centering of transform.

Example

1

2

```
.sample-class {
    transform-origin: 10px 10px;
}
```

1

2

3

```
.sample-class {
    transform-origin: right bottom;
}
```

Transition property

Transition shorthand

Transition shorthand has four following sub-properties, each of which can also be individually defined.

- transition-property
- transition-duration
- transition-timing-function
- transition-delay

You have to list the values without naming them individually. Values skipped will be assigned their default values.

Syntax

`transition: property duration timing-function delay;`

Example

`transition: margin-left 2s ease-in-out 0.5s;`

Animations and @keyframes

animation property:

Syntax

`animation: name duration timing-function delay iteration-count direction fill-mode play-state;`

Example

```
1  
2  
3  
  
.sample-class {  
  
    animation: none 2 ease 0.5 4 normal none running;  
  
}
```

The animation property is a shorthand for the sub-properties below:

```
1  
2  
3  
4  
5  
6  
7  
8  
  
animation-name
```

```
animation-duration  
animation-timing-function  
animation-delay  
animation-iteration-count  
animation-direction  
animation-fill-mode  
animation-play-state
```

The values not mentioned are given default values.
Animation-name property is used to tie-in the @keyframes rule.

@keyframes

Syntax

```
1  
2  
3  
4  
  
@keyframes mymove {  
    from {property: value}  
    to { property: value }  
}
```

Example

1

2

3

4

```
@keyframes animation-name {  
    from {bottom: 0px;}  
    to {bottom: 100px;}  
}
```

Percentage denotes the timing of the animation.

Alternative syntax

1

2

3

```
@keyframes animation-name {  
    /* declare actions here */  
}
```

Example

1

2

3

4

5

6

7

8

```
@keyframes animation-name {  
    0%, 100% {  
        background-color: blue;  
    }  
    50% {  
        background-color: green;  
    }  
}
```

Multiple animations

Works the same as regular animation, multiple rules can be set.

1

2

3

4

```
#some-class {  
    animation: animation-a 2s linear infinite alternate,  
            animation-b 3s ease infinite alternate;  
}
```

Completed

Preprocessors: sass, scss, Stylus

Preprocessors: sass, scss

Now that you have learned about different animation effects, let's explore the topic of preprocessors which can make the process of creating them easier. CSS preprocessors are special compilers used to create a CSS file that can be referenced by an HTML document. They are generally used to reduce the amount of CSS you need to write and allow you to re-use values across multiple rules. This will make re-using animations and effects much easier. And because preprocessors are an extension of CSS they'll help not just in animation but any CSS code. Let's learn a little more about them.

Preprocessors provide audit functionality on top of the CSS features already present. Some of the features of preprocessors include the option to create variables, loops, and if else statements.

Different preprocessors each have their own syntax and configurations for adding these features. Some of the most commonly used preprocessors include Sass, LESS, Stylus and PostCSS. The use of these preprocessors requires the installation of a compiler on top of your web server.

In the early days of CSS, the main problem developers faced was the difficulty of managing the code. The way CSS was designed made the code very long, messy and complex. It also made it difficult to troubleshoot. Preprocessors have their own scripting language that adds logical structures, automation properties, reusability and bloating of the code. You'll now explore some of the different preprocessors available.

SASS and SCSS

Syntactically Awesome Style Sheets (SASS) is a scripting language that CSS compiles and interprets into CSS. SCSS, which stands for Sassy CSS is the syntax for SASS and can be seen as an advanced version of both SASS and CSS. The difference between SASS and SCSS is best explained by the SASS documentation, which states:

"There are two syntaxes available for Sass. The first, known as SCSS (Sassy CSS) and used throughout this reference, is an extension of the syntax of CSS. This means that every valid CSS stylesheet is a valid SCSS file with the same meaning. This syntax is enhanced with the Sass features described below. Files using this syntax have the .scss extension.

The second and older syntax, known as the indented syntax (or sometimes just "Sass"), provides a more concise way of writing CSS. It uses indentation rather than brackets to indicate the nesting of selectors and newlines rather than semicolons to separate properties. Files using this syntax have the .sass extension."

This example highlights these differences.

Regular CSS:

2

3

4

```
body {  
  font: 100% Arial;  
  color: lightblue;  
}
```

This is the SCSS:

1

2

3

4

5

6

7

```
$font-stack: Arial;  
  
$primary-color: lightblue;  
  
body {  
  font: 100% $font-stack;  
  color: $primary-color;
```

```
}
```

SASS for the same block:

```
1  
2  
3  
4  
5  
6
```

```
$font-stack: Arial  
$primary-color: lightblue
```

```
body
```

```
  font: 100% $font-stack  
  color: $primary-color
```

The variables have been defined at the top with labels such as '\$font-stack' and '\$primary-color'. This is done with the '\$' suffix. The result for both will be the same, and it is not hard to imagine how much time this can save for the developer in complex code blocks where there are a number of occurrences of 'lightblue' color. These variables are placed at the top of the SCSS page. In the case of SASS, the variation has mainly removed the curly brackets and semi-colons from the code.

The nesting of selectors and separation of properties here is done by means of indentation. You should note that all this syntax is valid and will produce the same output.

For someone familiar with programming concepts, these preprocessors also allow the usage of math and other functions that can be utilized for adding rules conditionally.

Another important functionality in SASS is the use of directives. Let us explore a couple of directives called @mixin and @include.

Syntax

```
@mixin name { property: value; property: value; ... }
```

```
1
```

2

3

4

5

6

7

8

9

```
@mixin some-rules {  
  
    color: lightblue;  
  
    font-size: 25px;  
  
    font-weight: bold;  
  
}  
  
div {
```

```
    @include some-rules;  
  
}
```

There are two directives `@mixin` and `@include`, that are used here.

In the first step, you will add properties that you want to reuse inside `@mixin`.

In the second step, you use the second directive `@include` and add the mixin identifier that you have created using the `@mixin` directive.

Similar to these, there are a couple of other directives that are also used. `@import` allows the import of rules from another file, and `@extend` allows all the rules from a specific selector to be added inside another selector.

Stylus CSS

Now that you know how preprocessors behave let us explore one more of their type, called Stylus. If you continue to use the example above, the code for Stylus will look like this:

```
1  
2  
3  
  
body  
  
font 100% Arial  
  
color lightblue
```

It is not hard to miss the simplicity of the code without the colons, brackets or semicolons. But you should note that it is still allowed to use all of them in Stylus without any error. Similarly, you can also use '\$' or any other symbol before variables, but you are not 'required' to do so.

For someone unfamiliar with programming, functions are a block of self-contained code that consists of steps designed to accomplish and obtain the desired output. The preprocessors, as mentioned, allow the use of functions. Here is an example of this using Stylus.

```
1  
2  
3  
4  
5  
  
add(a, b)  
  
a + b  
  
  
div  
  
margin add(10px, 20px)
```

What is evident in the code above is that first, you have defined a function called ‘add’ and passed the variables ‘a’ and ‘b’ inside it. You added some functionality inside the function. In this case, you add the two values a and b with the ‘+’ or addition operator. Once you’ve done that, instead of assigning a value to the ‘margin’ property, you pass the function add with numeric pixel values passed to it. The output of this code will yield to a form ‘margin 30px’ after compilation.

These functions are useful when adding color gradation or creating advanced geometric shapes on your web page.

There are other features available for preprocessors too. And, just like any programming language, the space of CSS preprocessors is also competitive, and by no means are these the only options available.

Once you have gained an understanding of regular CSS, the usage of preprocessors should be explored. The use of preprocessors today is almost inescapable given the number of advanced features they provide which are not available in conventional CSS.

Completed

Additional resources

Here is a list of resources about animation and keyframes in HTML and CSS that may be helpful as you continue your learning journey.

[Transform and transition property](#)

[Detailed information about animation](#)

[Detailed information about @keyframes](#)

[More advanced examples for adding text effects \(1\)](#)

[More advanced examples for adding text effects \(2\)](#)

[More advanced examples for adding text effects \(3\)](#)

[Types of effects added to websites \(1\)](#)

[Types of effects added to websites \(2\)](#)

[Collection of examples using HTML/CSS](#)

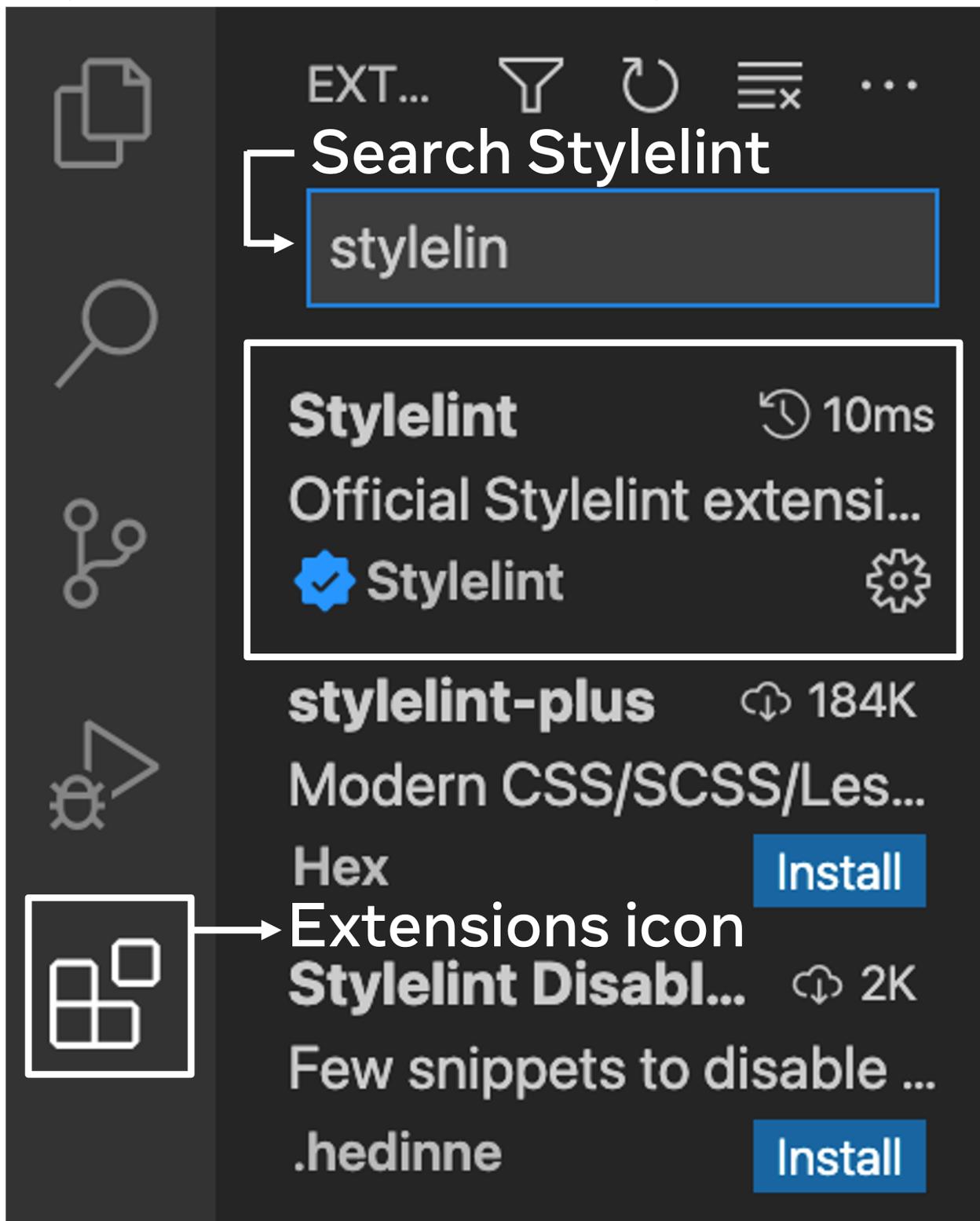
Completed

Installing a third-party Linter in VS Code

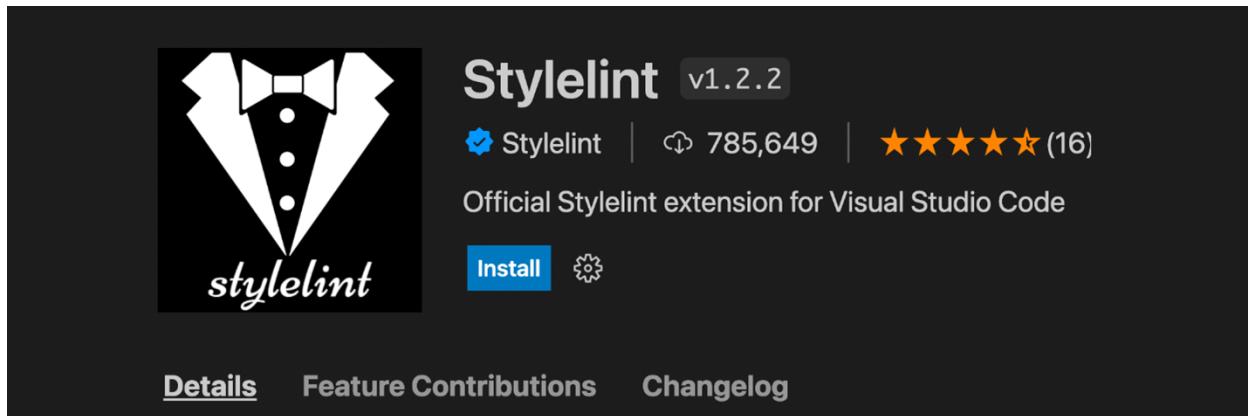
The extension you are going to use that will help you avoid errors is called Stylelint and can be found in the Extensions library of VSCode. Follow the steps below to install it and enable it to use on your CSS code.

Step 1: Install the extension

Inside your VSCode, click on the Extensions icon and search for Stylelint.



Click on the Install button in the Extension box that appears.



Step 2: Ensure it is enabled

Once installed, the button will be replaced by the 'Disable' and 'Uninstall' button.

If you see 'Enable' instead of 'Disable', click on 'Enable' and make sure the extension is enabled.

Step 3: Create and add a file called '.stylelintrc.json' to your project directory

The extension will look for this file inside any project folder you are working with.

Add the rules in JSON object format as below, these are the same as the ones you used in 'Handling errors'.

```
1
2
3
4
5
6
7
8
9
{
```

```

    "rules": {

        "alpha-value-notation": "number",

        "selector-type-case": "lower",

        "color-no-hex": true

    }

}

}

```

Additionally, you can explore their official website for other rules you may want to add.

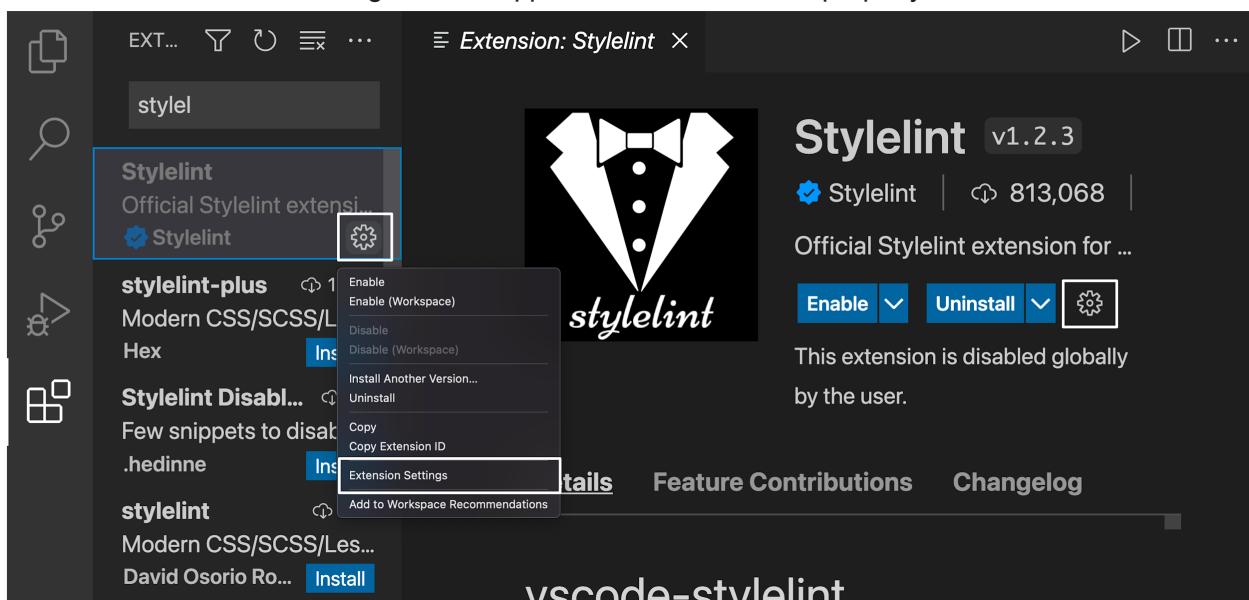
Step 4: Restart the VSCode

This will enable the extension completely.

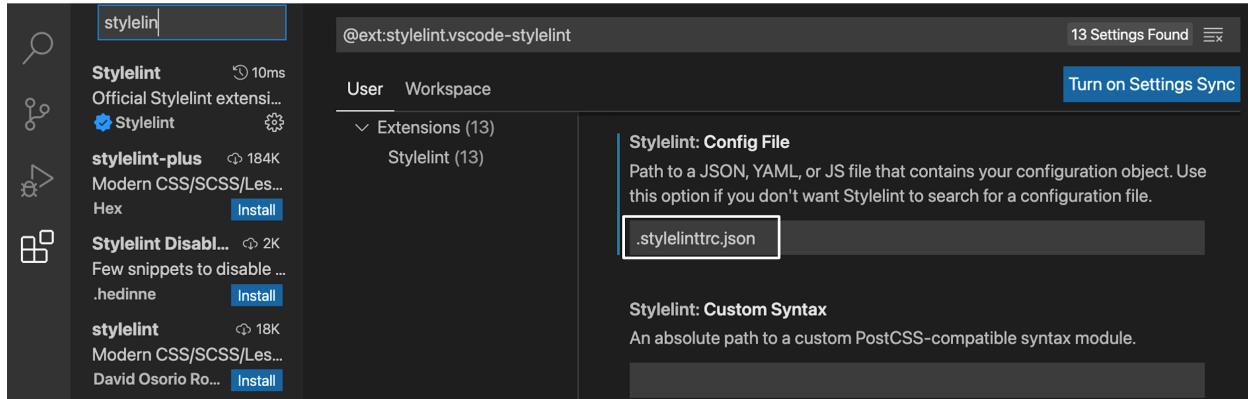
Step 5: Add `.stylelint.json` file inside the Extension settings

Go to the Stylelint extension again by clicking on the Extension icon and then searching for Stylelint. From directly next to the Stylelint title or next to the ‘Uninstall’ button, click on the ‘Settings’ icon and select ‘Extension settings’ from the drop-down list that appears.

Note that the Extension settings will not appear unless VSCode is properly reloaded.



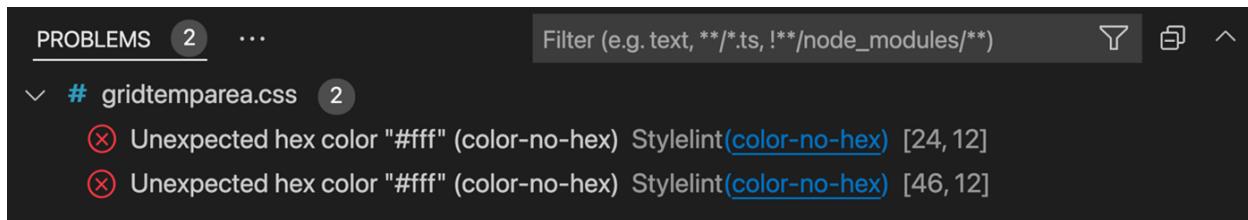
After clicking on the Extension settings, scroll down until you see the section Stylelint: Config File and add the value “.stylelintrc.json” under it. This is the same file name that you added inside your project folder.



Step 6: Look at the errors

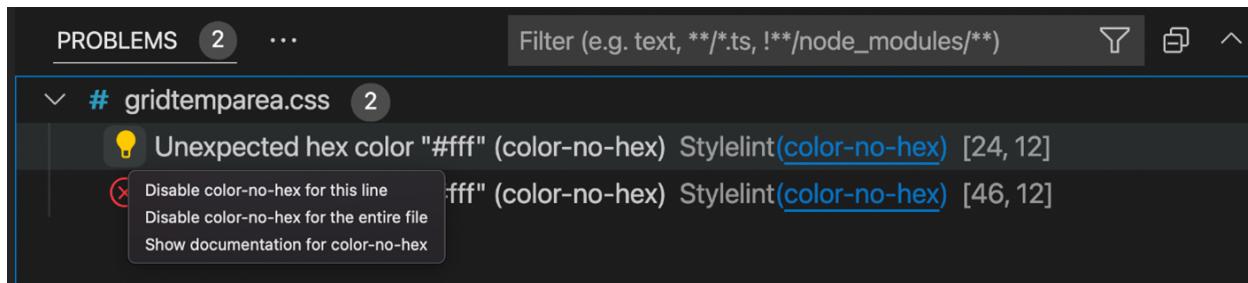
Now you can open a CSS file and see the changes from Linter at work.

Go to View > Problems in your Menu bar. This will open the ‘Problems’ tab at the bottom.



The errors found by the Linter will display like in the screenshot above. It shows information about the rule which flagged the error. Additionally, numbers in the bracket here are respectively the line and character number for the place the error was found.

Step 7: Light bulb



If you hover over any of the errors, a light-bulb icon will appear. If you click on it, a drop-down menu will appear. By selecting one of those options, you can explore the error that you have encountered in further detail.

And now you have installed and enabled Stylelint to use on your CSS code!

Note how specific rules can be set for specific projects and the Linter extension can be enabled or disabled inside your VSCode anytime. Specific lines can also be disabled for error-checks as in the

screenshot above where you can note the options for disabling the rule for the specific line or disabling a specific rule for the entire file. There are different rules that can be found as per the requirements of your project.

There are a number of good CSS Linters that are extensions or they can be accessed from their respective websites. Some of these are mentioned in the additional readings of this section.

Completed

Debugging the front-end

In this reading, you will learn about some of the fundamental tasks used to debug CSS. The scope of what's covered will match the topics already covered in this course.

You may have come across websites that have misaligned or overflowing text, broken web links or websites that take too much time to load. While the front-end and back-end both contribute to faulty web pages, visible styling is primarily concerned with the front-end.

CSS is a styling language, unlike conventional programming languages such as Python and Java, that has controls and follows a logical structure. This can make it difficult to find the exact root of the problem. Additionally, as you know, CSS does not flag errors, and most of the 'bugs' that you see in CSS are aesthetic in nature and need human intervention to solve. The task of debugging the front-end is more about experience than knowledge.

The first step in debugging CSS is to find the root of the issue and isolate the elements involved. The majority of CSS issues will be with the layout, such as:

- Content overflow from parent to child or container class
- Misplaced elements in relation to its container class
- Browser and device-related inconsistencies resulting in variable viewports

Isolation by reduced test case

One way to deal with a problem is to replicate your code and systematically remove any code unrelated to the HTML and CSS elements that could be the source of the problem. The code should be distilled down to the least amount of code possible, and only then are suitable changes made to get the results you want. Alternatively, you can enable rules one at a time to observe their impact on the displayed elements.

Items inside containers

Often, isolation will not work, as the problem is the result of the relative mapping of elements. For example, with the misconfigured width of an item inside a flex layout. It's important to check the use of suitable CSS properties in such cases. For a given item inside a grid, depending on the use case, width, grid-template-column, margin and padding can all be used to give spacing to an element. Additionally, you can also set different units that will all have their own behavior. In most cases, it

helps to be familiar with the unit of measurement in relation to the container type to avoid misconfigurations.

Relocating items

Similar to the isolation of elements, you can move a certain element to observe its behavior. Doing a comparison can often help you to understand the source of the problem.

The CSS compiler reads the elements from right to left. As an example, for a selector such as `div.alpha > p`, the element read first will be `p` before moving ‘outside’. When you change the position of the `p` from inside the `.alpha` class to some other position inside your code, it is easier to debug the source of the problem. This should be done on a case-specific basis.

Getting familiar with the box model

The box model is a very powerful source of information and can solve many issues with alignment. Using margin, padding and border is useful, but can be tricky and must be well understood.

Browser issues

Many times, the styling you have renders correctly in the IDE but misbehaves in a browser. That is because browsers have their own default CSS stylesheets called user-agent styles. While modern-day browsers are mostly compatible, you may encounter minor inconsistencies. Overriding the browser’s settings can be done with universal selectors, in such cases written at the top of the code, and will include properties such as ‘`margin: 0px;`’ to reset the margin values set by the browser by default.

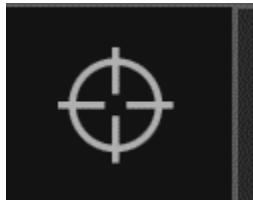
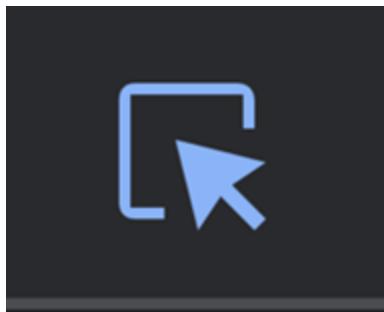
Dev tools

There are lots of user-friendly tools available today that can help debug CSS. However, the best tool you can use is the one provided by the browsers, called the developer tools, or dev tools. You can find these by right-clicking on a web page and selecting ‘Inspect Element’. Note how every browser has its own expression when it comes to the configuration options, but fundamentally they are similar. Browsers today are very powerful pieces of software. If you spend time exploring the options, you may not need any other additional tools or software for debugging CSS and other front-end languages.

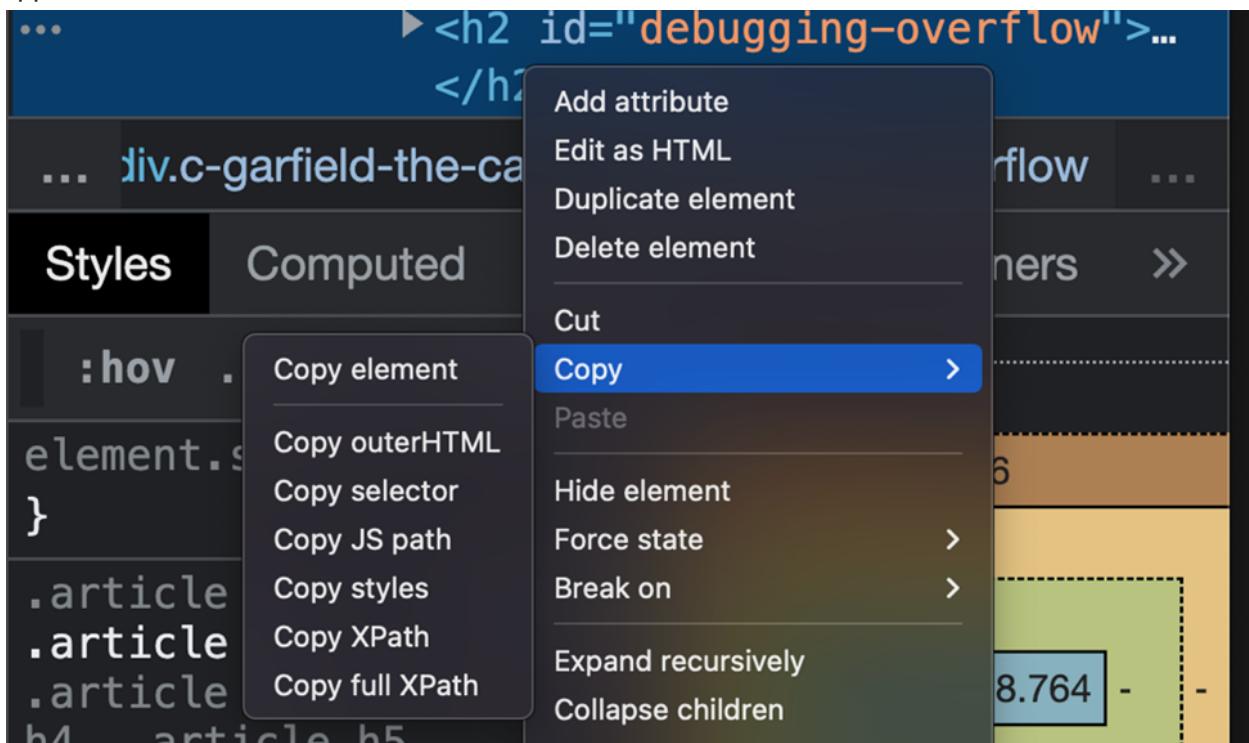
A couple of the important options you can find inside your browser include:

- Sources: Lists the filenames such as HTML, and CSS files used by the webpage that can be explored
- Elements: Scrolls through the code to select a specific element for exploration

Inside the Elements tab, on the right-hand side, you will find several options such as Computed layouts that will show the box model, Layouts that contain page and grid overlay options, and Font. You can select a specific element much more easily with the help of the ‘Element selection’ icon inside the dev tools. It enables the selection of specific elements on your web page. You can also access this option by hovering over a specific element on your page that will display its properties to you.

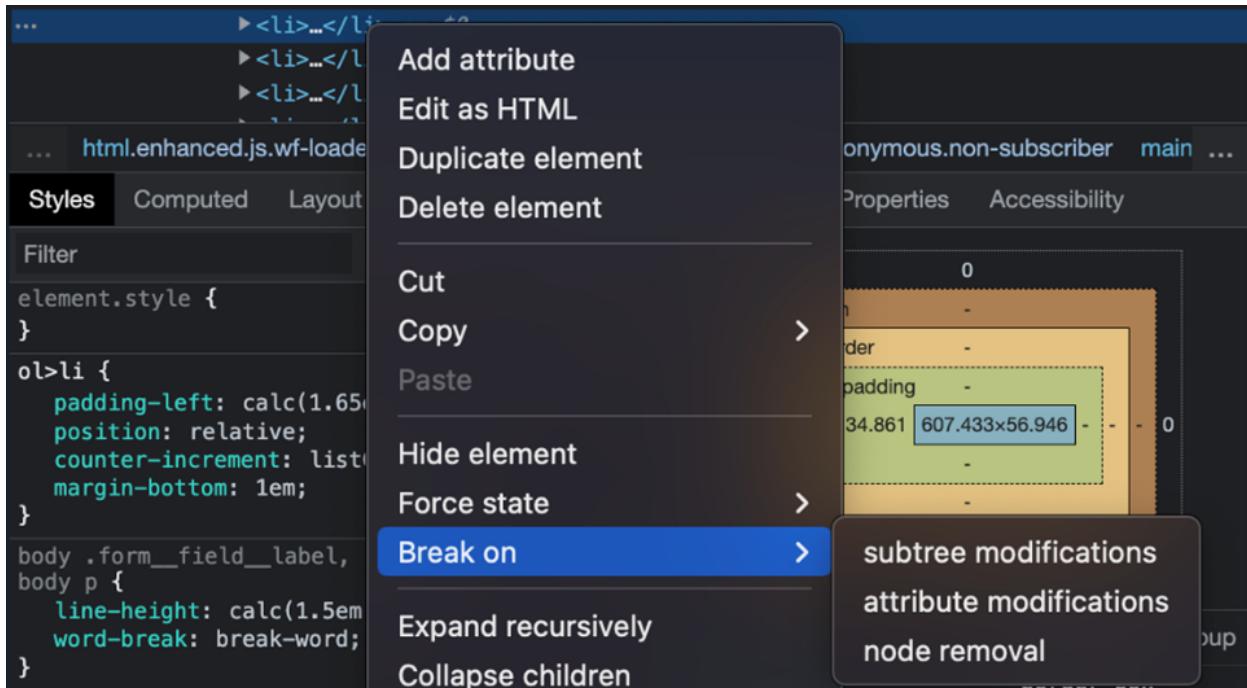


On selecting the desired element, one way to use the Elements tab for debugging is by right-clicking on that element, scrolling to 'Copy' and then selecting an option from the drop-down list that appears.



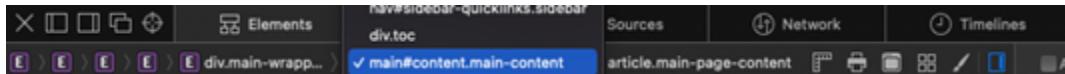
This way, you can explore the isolated code to find the problem.

Additionally, you can add the Breakpoint option that is more useful for interactive pages by selecting the 'Break on' option and selecting an option from the dropdown list that appears.



Nowadays, some browsers are providing options such as Cumulative Layout Shift (CLS) that helps determine the overall efficiency of a web page.

You can also bring up the element families by clicking on the horizontal bar:



One very useful feature is the ability to make changes in your code directly inside the browser. First, use the 'element selector' mentioned above to select some elements and then look for the '+' icon inside the dev tools. You can directly start adding relevant rules for that particular element which will immediately display changes on your web page. The changes you make can also be tracked from the 'Changes' tab. The live interactive nature of this feature greatly improves the experience of debugging.

'Console' is another feature that, although it is not that useful in CSS, will come in handy as you deal with active or dynamic web pages along your developer journey.

There are also responsive design modes in CSS that allow you to render your webpage to a specific browser or device. In addition to these, there are numerous ways in which you can explore and configure settings inside dev tools.

When it comes to designing and styling CSS, if you don't understand how it works, all issues will appear to be bugs. If you look at the fundamental structure of CSS, it consists of rule sets containing selectors and declaration blocks that contain properties and values. Micro-assessment of formatting and its validity can be done to troubleshoot the source of the problem. Practicing the creation of web pages and exploring the dev tools is the best way to get better at debugging and CSS in general. In this reading, you learned about some of the fundamental tasks used to debug CSS.

Completed

Additional resources

The following resources will be helpful as additional references in dealing with the different concepts related to the topics you have covered in this lesson.

[Website to check for browser-compatibility](#)

[Official website for Stylelint](#)

[Widely used linter website to check CSS code](#)

[CSS Validator to check accuracy of CSS code](#)

[Dealing with specificity issues](#)

[Beginner errors made in CSS](#)

[Knowledge-based mistakes to avoid in CSS](#)

[How to use Webkit in your CSS code](#)

[Commonly-occurring browser issues in CSS](#)

Completed

About the portfolio project

By completing the lessons in this course, you have acquired the necessary HTML and CSS skills and knowledge to develop a home page with a header, main content and footer. You will have to decide whether you are going to use a grid or flexbox layout for your home page. Your header should contain the client logo and the main content should be a large promotional banner with three columns with text and images below it. Your footer will have two columns: the first column should have a small logo and the second one should have copywriting information. When it comes to styling, you'll have to create the appropriate look and feel for your client's target audience.

Remember, you want to create intentional engagement!

To enable you to do all of this, the module covers the most essential features of your project. This includes: selecting your fictional client, working out your layout, positioning the elements, planning the user experience, creating themes and setting up your local development environment.

What is the purpose of the portfolio project?

The graded assessment will help you to establish which topics you have mastered, and which topics require further focus before you can complete the course. Ultimately, the graded assessment is designed to help you make sure that you are ready for the next course in this program. And even more, you get the opportunity to create another project for your portfolio.

How do I prepare for the graded portfolio project?

You have encountered exercises, knowledge checks, in-video questions and other assessments as you have progressed through the course. Nothing in the graded assessment will be outside what you have covered already, so you should be well placed to succeed.

What will I be doing?

The purpose of this graded assessment is to check your knowledge and understanding of the key learning objectives of this course. And what better way to do that than by applying your newly-learned skills in a practical way? You will develop a home page for one of four fictional clients. And you can read more about them in the [Subject selection](#) reading. The client you choose will determine what links, pictures and colors you will use. You are free to use your creativity but the home page needs to meet certain criteria regarding the following:

- Visual layout
- Semantic structure
- CSS layout
- CSS styling
- CSS effects

How will I get feedback?

This is a peer-review project which means that your home page will be evaluated by your fellow learners. Of course, this means that you will also need to give feedback to your peers and grade their home pages. In this way, you can learn from other learners' ideas and the issues that they may have encountered.

The rest of this lesson will guide you on how best to approach the development of your home page. Good luck!

Completed

Subject selection

For the graded assessment, you need to develop a home page for one of four fictional clients. As you read through the client personas included in this reading, think about what the ideal home page for each of these clients would look like.

How can you use what you've learned over the course? For instance, how will you structure the provided information with semantic HTML tags? You can even start to think about what media elements you could include.

And how can you create a unique user experience with CSS? Will a grid or basic flexbox serve the client best? What CSS selectors will you use to not only style your page but also create interactivity? And don't forget about effects! It's time to have fun and think about how you can apply what you learned about CSS animations.

The four client personas offer a great variety for you to choose from. They are:

- Retail: Lucky Shrub
- Professional services: Hair Day
- Restaurant: Little Lemon
- Luxury jewelry: Mangata and Gallo

Read on to find out more about them.

Retail: Lucky Shrub



Based in Tucson, Arizona, Lucky Shrub is a medium-sized garden design firm that specializes in garden design and creation, maintenance and landscaping. The company also runs a small plant nursery that sells indoor and outdoor plants, making them a one-stop shop for clients to "transform any space into an oasis you can be proud of".

Lucky Shrub was started by a husband and wife team, Jason and Maria, who share a long-time love for plants. Jason is the garden architect. He creates and oversees all designs while managing his team of landscapers. Maria manages all the marketing for the company and oversees the nursery. Download their logos here:

[Lucky-Shrub-Logos](#)

[ZIP File](#)

Professional services: Hair Day



Based in Madison, Wisconsin, Hair Day is a boutique hair salon that specializes in cut, color and styling. Hair Day also offers makeup and nail services, prides itself on its warm and relaxing atmosphere and is best known for edgy hair colors and on-trend cuts. The salon can only accommodate a small number of coloring appointments each week, so clients need to book far in advance. To keep clients inspired between appointments, the employees share makeup and nail tutorials on the Hair Day website.

Pria, the owner of Hair Day, earned her certificate as a colorist and immediately started specializing in edgy, custom coloring techniques. She wanted to create a "treat yourself" environment for her clients. Pria opened the salon with her good friend and talented hairdresser, Garry. They have a staff of seven: two stylists, three colorists, one manicurist and one make-up artist.

Download their logos here:

[Hair Day logos](#)

[ZIP File](#)

Restaurant: Little Lemon



L I T T L E L E M O N

Based in Chicago, Illinois, Little Lemon is a family-owned Mediterranean restaurant, focused on traditional recipes served with a modern twist. The chefs draw inspiration from Italian, Greek, and Turkish culture and have a menu of 12–15 items that they rotate seasonally. The restaurant has a

rustic and relaxed atmosphere with moderate prices, making it a popular place for a meal any time of the day.

Little Lemon is owned by two Italian brothers, Mario and Adrian, who moved to the United States to pursue their shared dream of owning a restaurant. To craft the menu, Mario relies on family recipes and his experience as a chef in Italy. Adrian does all the marketing for the restaurant and led the effort to expand the menu beyond classic Italian to incorporate additional cuisines from the Mediterranean region.

Download their logos here:

[Little-Lemon-logos](#)

[ZIP File](#)

Luxury Jewelry: Mangata and Gallo

MANGATA & GALLO

Mangata and Gallo is a jewelry store that specializes in special occasions like engagements, weddings and anniversaries. The jewelry company primarily operates online and has a small storefront in Austin, Texas with an atelier attached for browsing. Mangata & Gallo's selection of jewelry is known for its high-quality and classic look. The owner, an Austin local, is well known for her jewelry designs.

Mariana is the owner and the lead designer of Mangata and Gallo. After graduating from design school with a specialization in diamond cutting and metal smithing, Mariana opened a store in her hometown, Austin and started to grow her business online. Mariana has always managed every aspect of the business, from jewelry design to marketing to e-commerce. However, she recently hired several artisans to help craft her designs and a young employee to help manage the company website and social media accounts.

Download their logos here:

[Mangata and Gallo logos](#)

[ZIP File](#)

Completed

Setting up your local development environment

You have to use Visual Studio Code to complete your graded assessment. You have two options to do this.

Option 1: Use Visual Studio Code in-browser with Coursera Labs

You can access the Visual Studio Code environment through the “Ungraded Lab Project sandbox” included in this lesson. Take note that the Project Sandbox only allows you to work for one hour at a time. Make sure you download your files before exiting the UGL. To work on your project again later, you can simply open the HTML and CSS files on your local machine and copy and paste the code again into the template files in the Project Sandbox. Remember to download the edited versions again at the end of the session.

To submit your project, you need to download the HTML and CSS files to your local machine and save them inside a project folder. Zip the project folder and upload it in the "My submission" area in the Peer review assessment later in this lesson.

Option 2: Work on your local device

You can also choose to complete your graded assessment on your local machine if you prefer. This will require a few steps of set up in advance.

First, you need to download the free IDE from Microsoft's website -
<https://code.visualstudio.com/download>.

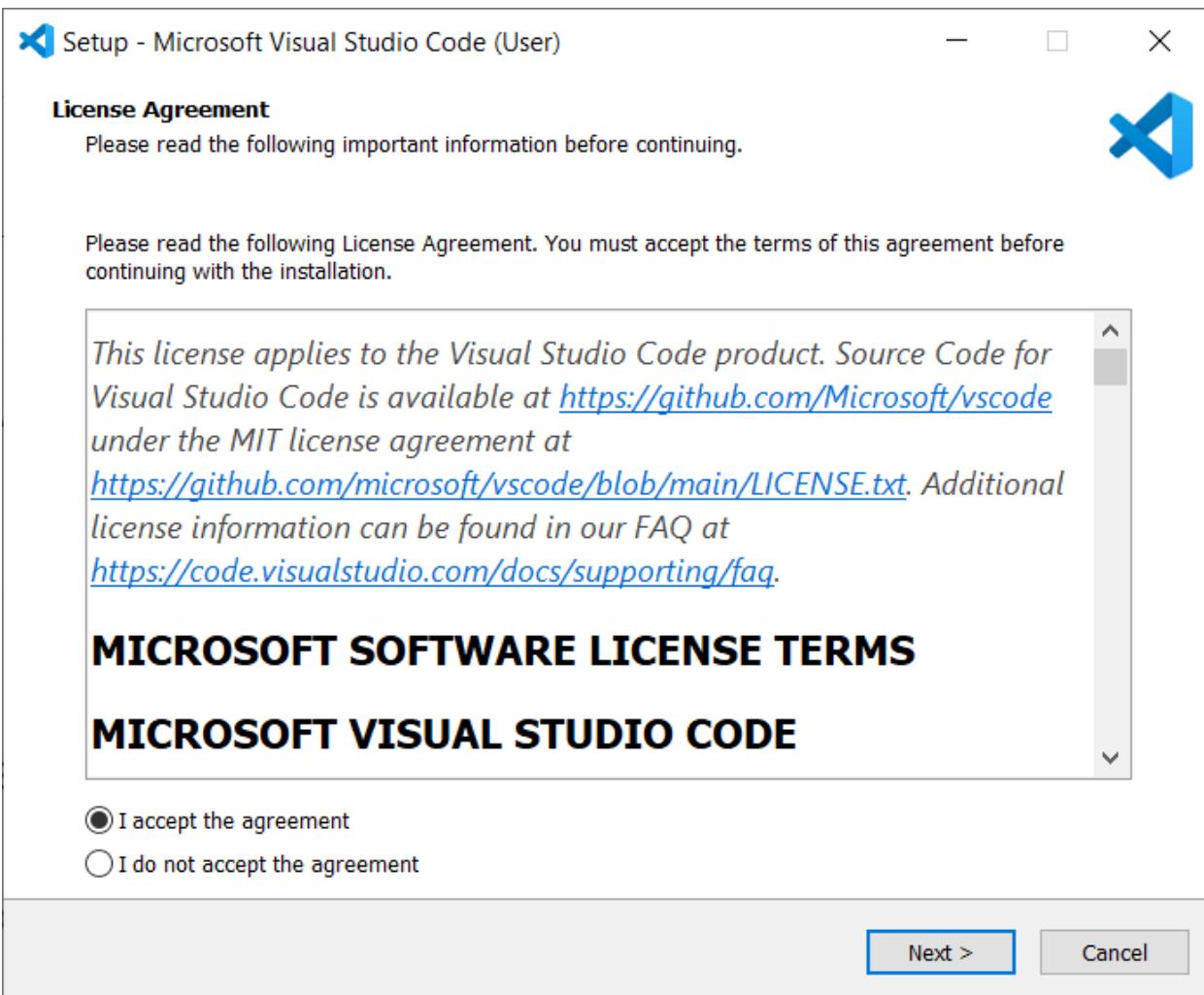
Select the download based on your operating system.

Windows

Step 1: Download the Windows installer.

Step 2: Open the file to install it once the download is complete.

Step 3: Review and accept the license agreement, then click Next.



Step 4: Keep the default value when prompted for the destination location and click next.



Setup - Microsoft Visual Studio Code (User)



Select Destination Location

Where should Visual Studio Code be installed?



Setup will install Visual Studio Code into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

 Browse...

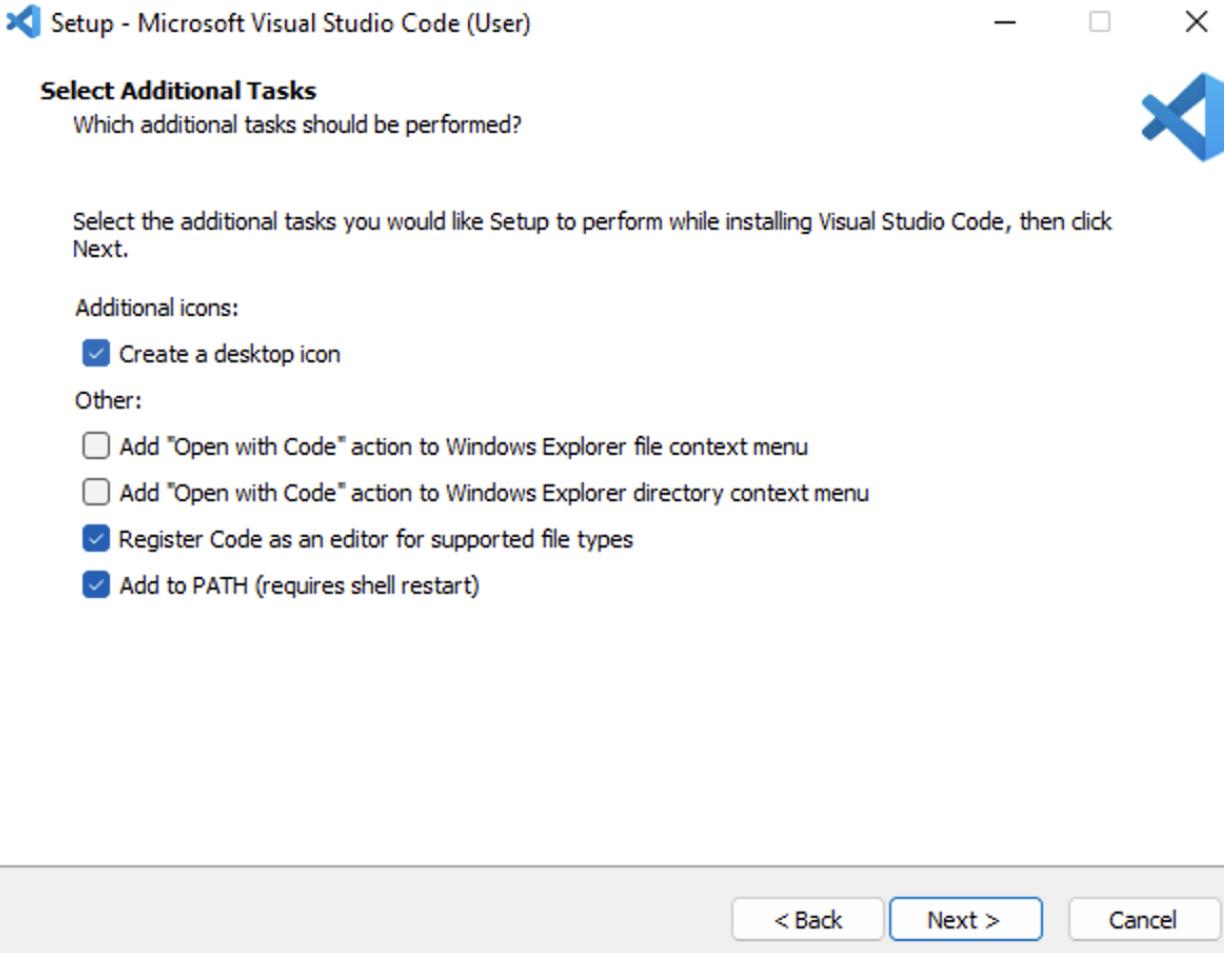
At least 292.3 MB of free disk space is required.

< Back

Next >

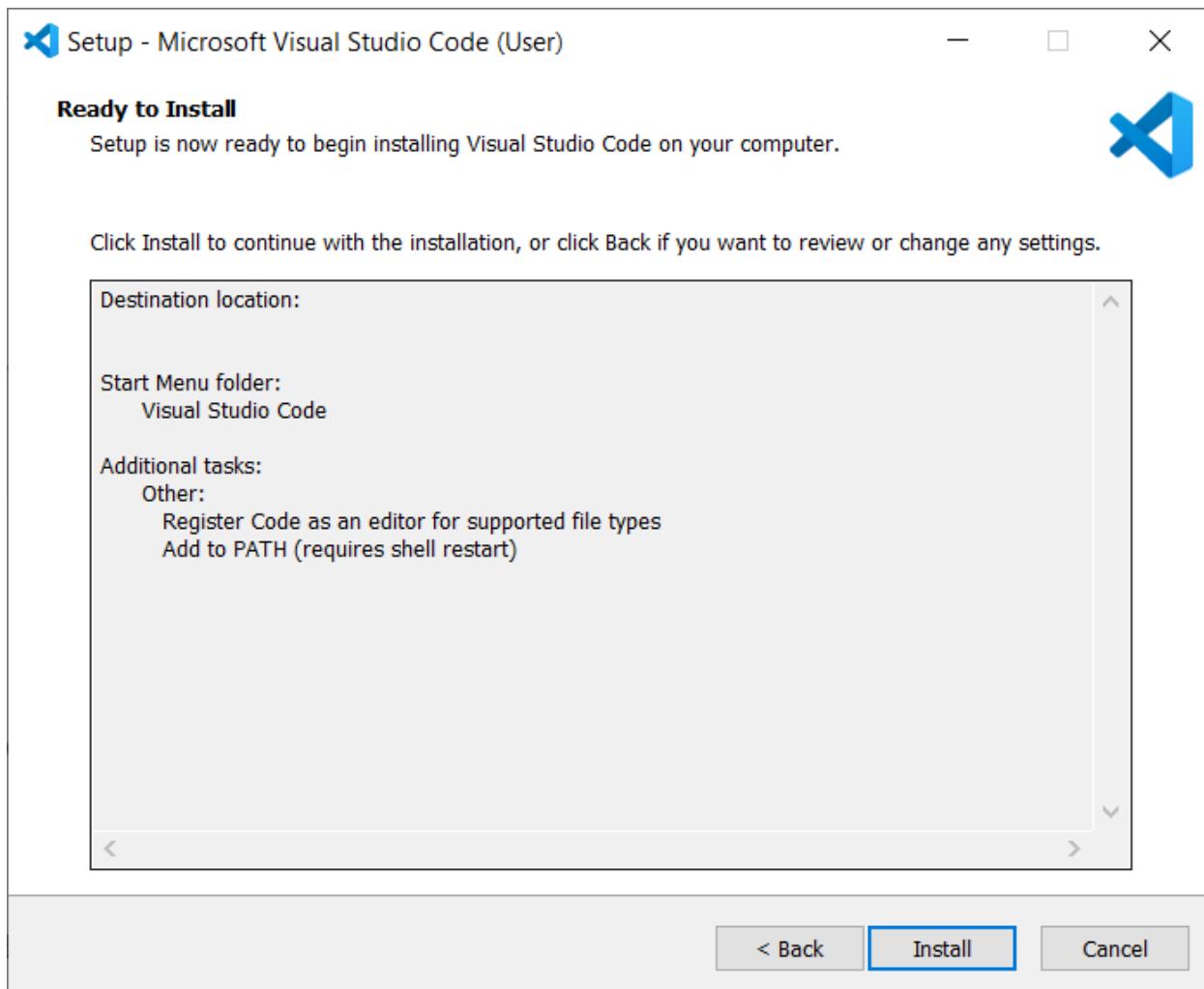
Cancel

Step 5: On the additional tasks view, make sure that **Add to PATH** is selected.



Step 6: Click next.

Step 7: Click install when the ready to install page appears.



Step 8: Click finish once completed, and the application will launch.

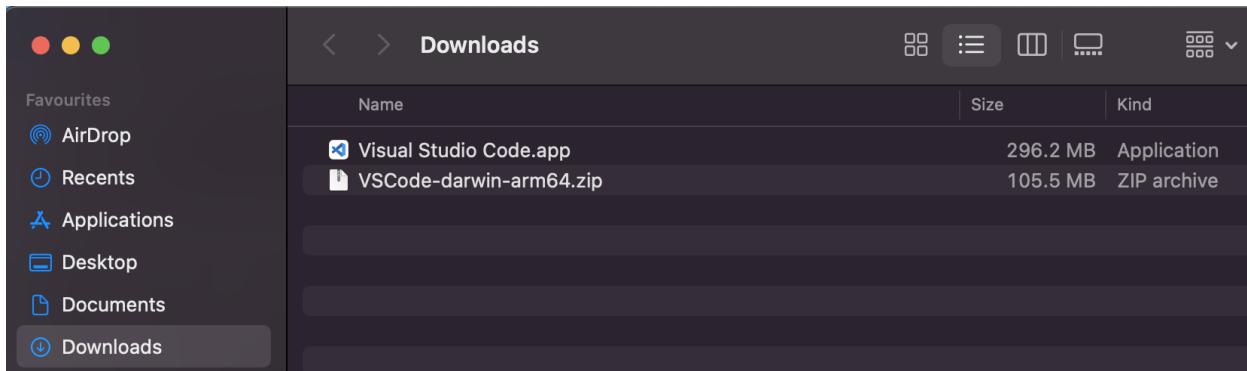
Mac

Step 1: Download the application based on the chipset you have. M1 macs use Apple Silicon, and older Macs use Intel. If you are not sure, choose the Universal option.

Step 2: Go to your Downloads folder once the download is complete.

Step 3: Double-click the zip file to extract the contents.

Step 4: Drag and drop the .app file to the application link in Finder below.



Step 5: Open the app.

Linux

Please refer to the [official Linux installation guide](#) for Visual Studio Code.

Selecting a working directory

Now that you have Visual Studio Code set up create a folder on your device that you'll use to do course exercises.

Open Visual Studio Code, go to `File` and select `Open Folder`. Using the file browser, select the folder you just created.

Congratulations, you're set up now to begin writing some code.

Completed

Creating themes

The combination of colors used in the design of a website is called the color scheme. Colors are one of the most important components of any website today because it sets the tone for the viewer. For the optimal use of colors, web developers make use of color theory and color schemes.

Color schemes

Fundamentally all colors are a combination of the three colors: red, yellow, and blue. These are called the primary colors. Secondary colors like orange, purple and green are a combination of two primary colors. Finally, there are six tertiary colors formed from a combination of the three primaries with secondary colors. By adding more black or more white you can create lighter or darker tints of each color. All of these colors and tints make up a color wheel of 12 colors. This color wheel is the foundation for any color scheme.

Depending on the relative positions of colors on the wheel, there are seven commonly identified color schemes according to color theory that can be used for maximum appeal. These are:

- Monochromatic

- Analogous
- Complementary
- Split complimentary
- Triadic
- Square and
- Rectangle

The different combinations are represented in the image below.

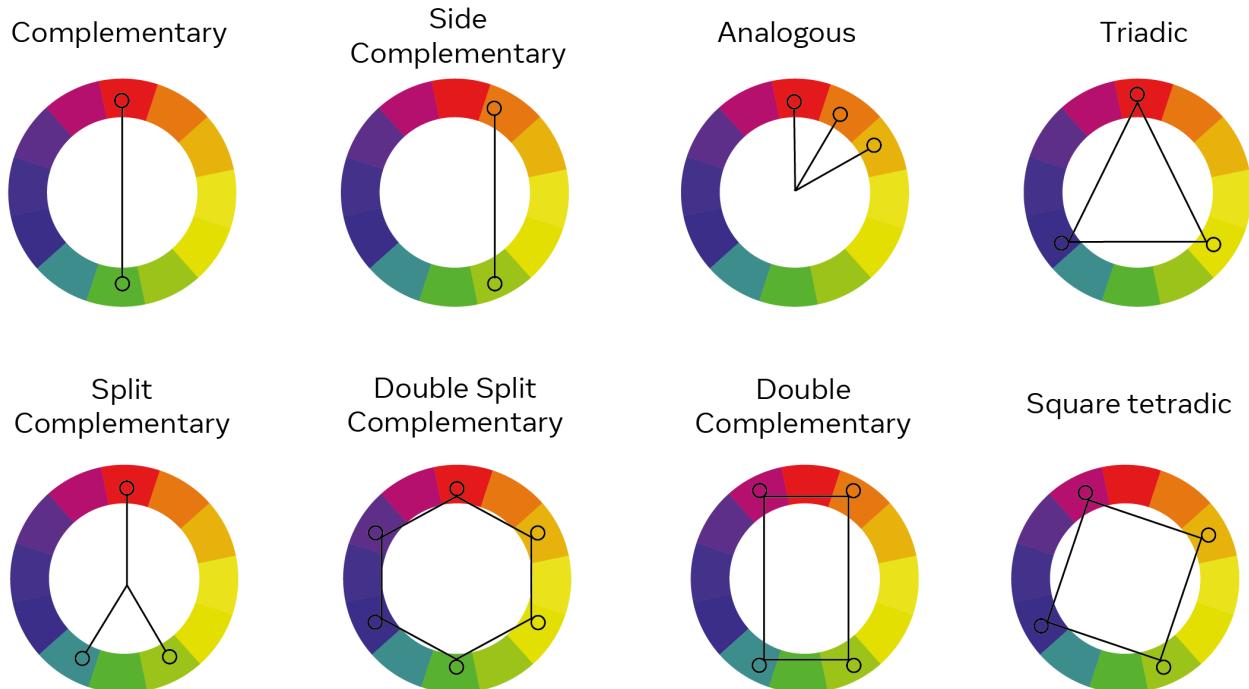


Image source: <https://www.moving.com/tips/how-to-choose-a-color-scheme-for-your-home/>

Using the color schemes

Each of the different color schemes are used with the purpose of enhancing a website. While you can pick any of the different schemes, a practiced UX designer will consider whether a specific type of color scheme is more suitable for certain websites than others. For example, complementary colors are well-suited for websites that will have bars and charts. It also depends on several other factors such as the number of colors that you are going to need for your webpage, the topic and domain of the webpage, user demographics of the readers and so on. For example, an informational website for medical services will typically have a white background. Other than the color schemes mentioned, even achromatic themes with only black and white color can be appealing if styling and design is well presented.

Factors to consider for themes and colors

While color schemes serve as a guide for picking website colors, there are some factors that must be considered in the process. The user experience must be of the utmost priority as a good webpage must be able to sustain the attention of the reader without overwhelming them. A good webpage will have a good balance of information and design elements. An unbalanced web design

can lead to what we call as cognitive overload. Cognitive overload is when too much information or activity is presented to the user which may be detrimental to the user experience.

There is a field of research in psychology that focuses on the effects color has on the mood of a user. Nature-inspired color tones are almost universally seen to be more pleasant and add aesthetic value. As people are increasingly spending time on the internet, a good theme may make or break the appeal of a webpage. Often, it is a good idea to keep a palette of more than one color combination on hand so you can experiment with your webpage.

A color theme on a website also plays a role in other aspects beyond the aesthetics and help in shaping the user experience of the website. A simple theme with appropriate color combinations can help a user navigate the website by creating a visual hierarchy. If for example, a user observes a specific color scheme for the subheadings, they will be able to understand where to look for certain information on the web page. This means you can avoid excessive design elements such as line-breaks and grids where they are not adding to the appeal of the web page.

While the experience of choosing the right colors for your website is a process of trial and error, inspiration can be drawn from a webpage that has held a personal appeal. Color theory is an important aspect of the user experience domain and plays an important role in website design so don't be afraid to experiment with different color schemes.

Completed

Next steps

Congratulations! You've taken another step toward improving your knowledge, skills, and qualifications. By completing this course you've acquired and practiced the core skills needed to write and test HTML and CSS code. The next course is React Basics and in it, you'll develop a working knowledge of React. React is a powerful JavaScript library that you can use to build user interfaces for web and mobile applications (apps). In this course, you will explore the fundamental concepts that underpin the React library and learn the basic skills required to build a simple, fast, and scalable app. Be sure to take the next course. It's your chance to gain further insight into the world of software development.

Completed

Courses5 -React Basics

Course syllabus for React Basics

By the end of this reading, you will have learned about the scope of things you will cover in this course.

Prerequisites

To take this course, you should understand the basics of HTML, CSS, and JavaScript. Additionally, it always helps to have a can-do attitude!

Course content

This course is an introduction to React development. You'll learn enough basic concepts to empower you to build simple user interfaces in React.

This course consists of four modules. They cover the following topics.

Module 1: Anatomy of React

In this introductory module, you'll learn about what React is and where it is used. You'll also learn how to set up your coding environment so that you have as productive a learning experience as possible. So, the purpose of this module is to understand the what and the why, and to get set up for the modules that follow.

Components are one of the foundations of React. In React, everything revolves around components. You'll learn how to build components, how to structure and customize your React projects, and how to compose layouts by importing components into other components.

You'll learn about passing data from one component to another. You'll learn about JSX syntax in React and how to use it to structure and style your components.

By the end of this module you will be able to:

- Explain the concepts behind React and component architecture.
- Describe how to use assets within an app to apply styling and functional components.
- Create a component to service a specific purpose.
- Create a folder and demonstrate how to create and import files within that folder.
- Use and manipulate props and components to effect visual results.

Module 2: Data and State

The second module of this course deals with working with events and errors in React. You'll learn how events work and how you can handle them in React. Handling events can sometimes get a bit tricky, so you'll also learn about dealing with errors related to events in React.

By the end of this module you will be able to:

- Use common methods to manage state in React.
- Detail the concept and nature of state and state change.
- Describe the hierarchical flow of data in React.
- Describe how data flows in both stateful and stateless components.
- Use an event to dynamically change content on a web page.
- Describe some common errors associated with events and the syntax required to handle them.

Module 3: Navigation Updating and Assets in React

In this module, you'll learn about routing and navigation in React. You'll learn how to render partial views and how to update routes in your React apps. You'll understand how assets are used, bundled and embedded.

By the end of this module you will be able to:

- Use media assets, such as audio and video, with React.
- Demonstrate how to manipulate image assets using reference paths.
- Explain the folder structure of a React project in terms of embedded or referenced assets.
- Demonstrate the conditional implementation and rendering of multiple components.
- Create and implement a route in the form of a navbar.
- Describe navigation design in React, with a focus on single and multi-page navigation.

Module 4: Portfolio Mini-Project (Calculator App)

This module is focused on a practical mini project of building a calculator app in React. Upon completing this module, you'll have coded your own mini project in React, as a starting point for building your React portfolio.

You have now learned about the scope of things you will cover in this course.

By the end of this module you will be able to:

- Synthesize the skills from this course to create and style a React component.
- Reflect on this course's content and on the learning path that lies ahead.

Completed

How to be successful in this course

Taking an online course can be overwhelming. How do you learn at your own pace and successfully achieve your goals?

Here are some general tips that can help you stay focused and on track.

Set daily goals for studying

Ask yourself what you hope to accomplish in your course each day. Setting a clear goal can help you stay motivated and beat procrastination. The goal should be specific and easy to measure, such as "I'll watch all the videos in Module 2 and complete the first programming assignment". And don't forget to reward yourself when you make progress towards your goal!

Create a dedicated study space

It's easier to recall information if you're in the same place where you first learned it, so having a dedicated space at home to take online courses can make your learning more effective. Remove any distractions from the space and if possible, make it separate from your bed or sofa. A clear distinction between where you study and where you take breaks can help you focus.

Schedule time to study on your calendar

Open your calendar and choose a predictable, reliable time that you can dedicate to watching lectures and completing assignments. This helps ensure that your courses won't become the last thing on your to-do list.

Tip: You can add deadlines for a Coursera course to your Google calendar, Apple calendar, or another calendar app.

Keep yourself accountable

Tell your friends about the courses you're taking, post achievements to your social media accounts or blog about your homework assignments. Having a community and support network of friends and family to cheer you on makes a difference!

Actively take notes

Taking notes can promote active thinking, boost comprehension and extend your attention span. It's a good strategy to internalize knowledge whether you're learning online or in the classroom. So, grab a notebook or find a digital app that works best for you and start synthesizing key points.

Tip: While watching a lecture on Coursera, you can click the 'Save Note' button below the video to save a screenshot to your course notes and add your own comments.

Join the discussion

Course discussion forums are a great place to ask questions about assignments, discuss topics, share resources and make friends. Our research shows that learners who participate in the discussion forums are 37% more likely to complete a course. So make a post today!

Do one thing at a time

Multitasking is less productive than focusing on a single task at a time. Researchers from Stanford University found that "People who are regularly bombarded with several streams of electronic information cannot pay attention, recall information or switch from one job to another as well as those who complete one task at a time." Stay focused on one thing at a time. You'll absorb more information and complete assignments with greater productivity and ease than if you were trying to do many things at once.

Take breaks

Resting your brain after learning is critical to high performance. If you find yourself working on a challenging problem without much progress for an hour, take a break. Walking outside, taking a shower or talking with a friend can help you to re-energize and even give you new ideas on how to tackle the project.

Your learning journey starts now!

While preparing for the module quiz or working on achieving your learning goals you're encouraged to:

- Work through each lesson in the learning pathway. Try not to skip any activities or lessons unless you are certain that you already know this information well enough to move ahead.

- Take the opportunity to go back and watch a video or read all the information provided before moving on to the next lesson or module.
- Complete all the knowledge and module quizzes and exercises.
- Read the feedback carefully when answering quizzes, as this will help you to reinforce what you are learning.
- Make use of the practical learning environment provided by the exercises. You can gain substantial reinforcement of your learning through the step-by-step application of your skills.

Completed

Before you learn React

Do you know the fundamentals of HTML, CSS and JavaScript? Perhaps you learned about these technologies from another course. Either way, a quick summary will be useful so let's explore some fundamental HTML, CSS and JavaScript principles and practices.

In this reading, let's take a practical approach, and revisit some of the development techniques you'll need to be comfortable with before learning React.

To get the most out of this course on React basics, you should first understand the fundamental methods and concepts of JavaScript. Otherwise, you may feel like you're a child learning to run before you can walk. React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called "components".

React apps are built using modern JavaScript features, which are commonly known as ES6. Developers use React to develop Single Page Applications. And you can also develop mobile applications with React Native.

As an aspiring developer, you may opt for a 'learn as you go approach' regarding JavaScript and React. But this may not help your productivity and even at times frustrate you. This is because you may confuse code or functionality that is plain JavaScript, or code that is React.

For example, with a solid foundational knowledge of JavaScript, you can quickly identify code that is JavaScript ES6 and code that is React. And throughout this course, there will be help for you along the way with some friendly reminders.

Also, keep in mind that you are using React to build user interfaces which also include HTML and CSS code.

So let's begin with the fundamental HTML knowledge needed to learn React.

HTML

Recall that HTML is used to describe the structure of Web pages. Developers use HTML elements with their opening and closing tags to "mark up" an HTML document.

These elements form the structure of a web page and describe what to display to the web browser. When it comes to HTML it's important to know about:

1. The purpose of HTML in the web browser,
2. the use of HTML tags and correct syntax,
3. and how HTML elements are used in a web document.

Another important concept to know about when you're talking about HTML is the Document Object Model, or DOM.

Users need to be able to interact with elements on a web page. This means that an HTML document must be represented in a way that JavaScript code can query and update it. And that's the function of the DOM. It's a model of the objects in your HTML file.

And web developers interact with the DOM through JavaScript to update content, set up events and animate HTML elements.

Before you learn React, it's advisable that you are comfortable with the following HTML tags and concepts

Layout & Style

- `<html>`
- `<head>`
- `<body>`
- `<div>`

Text formatting & lists

- `<h1>...<h6>`
- `<p>`
- ``
- `<i>`

Images and links

- ``
- ``

Linking and Meta

- `<link>`
- `<title>`
- `<meta>`

Semantic

- `<header>`

CSS

CSS (Cascading Style Sheets) is the code that you use to style HTML. You need to be familiar with basic CSS concepts before you start learning React. This is because you will need to style your React components as well, and basic CSS knowledge will help your learning journey.

Before you learn React, make sure you are comfortable with these CSS styling options:

- Font styling (font size, font color, etc.)
- Flex Box Layout (Layout of items using CSS Flex Box Layout)
- CSS Selectors
- Position, Padding, Margins and Display

- Colors, Background and Icons

You can refresh your knowledge of HTML and CSS in the Meta course titled: [Introduction to Front-End Development](#)

JavaScript fundamentals and ES6

React is completely written in JavaScript and uses the more modern version of JavaScript which is ES6. While learning React, you should already know JavaScript fundamentals.

JavaScript is the programming language and React is a JavaScript UI library. This means the first step is to be proficient at JavaScript.

Here are some of the JavaScript topics that you need to be comfortable with before you begin your journey learning React.

- Data types
- Using var, let and const
- Conditionals and Loops
- Using objects, arrays and functions
- ES6 Arrow functions
- In-built functions such as map(), forEach() and promises.
- Destructuring Arrays and Objects
- Error Handling

Package Manager (Node + npm)

React is a UI library, and you will encounter that many times you will need to add other packages to your React application. A package in JavaScript contains all the files needed for a module. To install these packages effectively and manage their dependencies you can use a package manager like NPM (Node Package Manager).

You can install npm by installing Node.js, which will then automatically install npm.

You need to be comfortable with using npm as your package manager, since you will be using npm to install packages within your React application. Make sure you are aware of how to do the following with npm before you get started on this course.

- Installation command to install npm modules in your project
- Installing a package as a dev dependency
- Start command
- Updating npm version
- Navigating around the package.json file

Once you have become confident with these skills, you'll be in a better position to learn and apply React concepts and prepare yourself for development of React apps.

To refresh your knowledge of JavaScript and the basics of Node and npm, please visit Meta course titled: [Programming with JavaScript](#).

Completed

JavaScript modules, imports - exports

Before you start creating the next great app, let's explore a little more about modules.

Modules can help you to save and access your code in a more structured way, and in this reading, you'll learn about some foundational concepts of working with JavaScript modules.

This knowledge is crucial in order to understand the syntax and the logic behind how the example React apps in this course are put together.

This reading will cover the three main concepts:

1. JavaScript modules
2. Module exports
3. Module imports

JavaScript Modules

In JavaScript, a module is simply a file.

The purpose of a module is to have more modular code, where you can work with smaller files, and import and export them so that the apps you build are more customizable and have more composable parts.

A module can be as simple as a single function in a separate file.

Consider the following function declaration:

```
1  
2  
3  
  
function addTwo(a, b) {  
  
    console.log(a + b);  
  
}
```

Say that you have a file named **addTwo.js** that contains only the above code.

How would you make this file a JavaScript module?

All that you would need to do to make it a JavaScript module is use the export syntax.

Module Exports

There is more than one way to export a module in JavaScript.

While all the various syntactical differences are not listed, here are a few examples that will cover all the ways that the importing and exporting of JavaScript modules will be done in this course.

In general, there are two ways to export modules in JavaScript:

1. Using default exports
2. Using named exports

Default Exports

You can have **one default export** per JavaScript module.

Using the above **addTwo.js** file as an example, here are two ways to perform a default export:

```
1  export default function addTwo(a, b) {  
2      console.log(a + b);  
3  }  
4  
5
```

So, in the above example, you're adding the `export default` keywords in front of the `addTwo` function declaration.

Here's an alternative syntax:

```
1  function addTwo(a, b) {  
2      console.log(a + b);  
3  }  
4  
5  
6  export default addTwo;
```

Named Exports

Named exports are a way to export only certain parts of a given JavaScript file.

In contrast with default exports, you can export as many items from any JavaScript file as you want.

In other words, there can be only one default export, but as many named exports as you want.

For example:

```
1  
2  
3  
4  
5  
6  
7  
  
function addTwo(a, b) {  
  
    console.log(a + b);  
  
}  
  
  
  
function addThree(a + b + c) {  
  
    console.log(a + b + c);  
  
}
```

If you want to export both the `addTwo` and the `addThree` functions as named exports, one way to do it would be the following:

1

2

3

4

5

6

7

```
export function addTwo(a, b) {  
  
    console.log(a + b);  
  
}
```

```
export function addThree(a + b + c) {  
  
    console.log(a + b + c);  
  
}
```

Here's another way you could do it:

1

2

3

4

5

6

7

8

9

```
function addTwo(a, b) {
```

```
        console.log(a + b);  
    }  
  
function addThree(a + b + c) {  
  
    console.log(a + b + c);  
  
}  
  
export { addTwo, addThree };
```

Importing Modules

Just like when exporting modules in JavaScript, there are several ways to import them. The exact syntax depends on how the module was exported.

Say that you have two modules in a folder.

The first module is `addTwo.js` and the second module is `mathObj.js`.

You want to import the `addTwo` module into the `mathOperations` module.

You want to import the `addTwo.js` module into the `MathOperations.js` module.

Importing a Module that was Exported as Default

Consider the previous example of exporting the `addTwo` function as a default module:

1

2

3

4

5

6

```
// addTwo.js module:
```

```
    console.log(a + b);  
}  
  
export default addTwo;
```

To import it into the **mathOperations.js** module, you could use the following syntax:

```
1  
2  
3  
  
import addTwo from "./addTwo";  
  
// the rest of the mathOperations.js code goes here
```

So, you could start this import with the `import` keyword, then the name under which you'll use this imported code inside the **mathOperations.js** file. You would then type the keyword `from`, and finally the location of the file, *without the .js extension*.

Contrast the above import of the default `addTwo` `export` with the different import syntax if the `addTwo` function was instead a named export:

```
1  
2  
3  
  
import { addTwo } from "./addTwo";  
  
// the rest of the mathOperations.js code goes here
```

Conclusion

In this reading, you've learned about the very basics of what modules are in JavaScript, why they are used and how they get exported and imported.

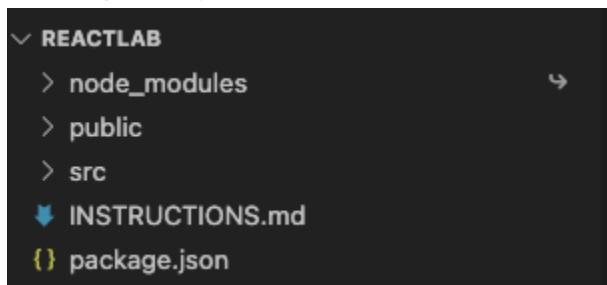
The examples you've seen here are the core of how you'll deal with imports and exports of various modules in the example React apps on this course.

However, please note that there are many more caveats, rules, and implementations of working with modules in JavaScript. The examples given in this reading are there just to make it easier to comprehend what is happening in React apps that you'll be building in this course. The intent of this reading was just to get you familiar with the most common syntax used - not as a comprehensive overview of modules in JavaScript.

Completed

Working with Labs in this course

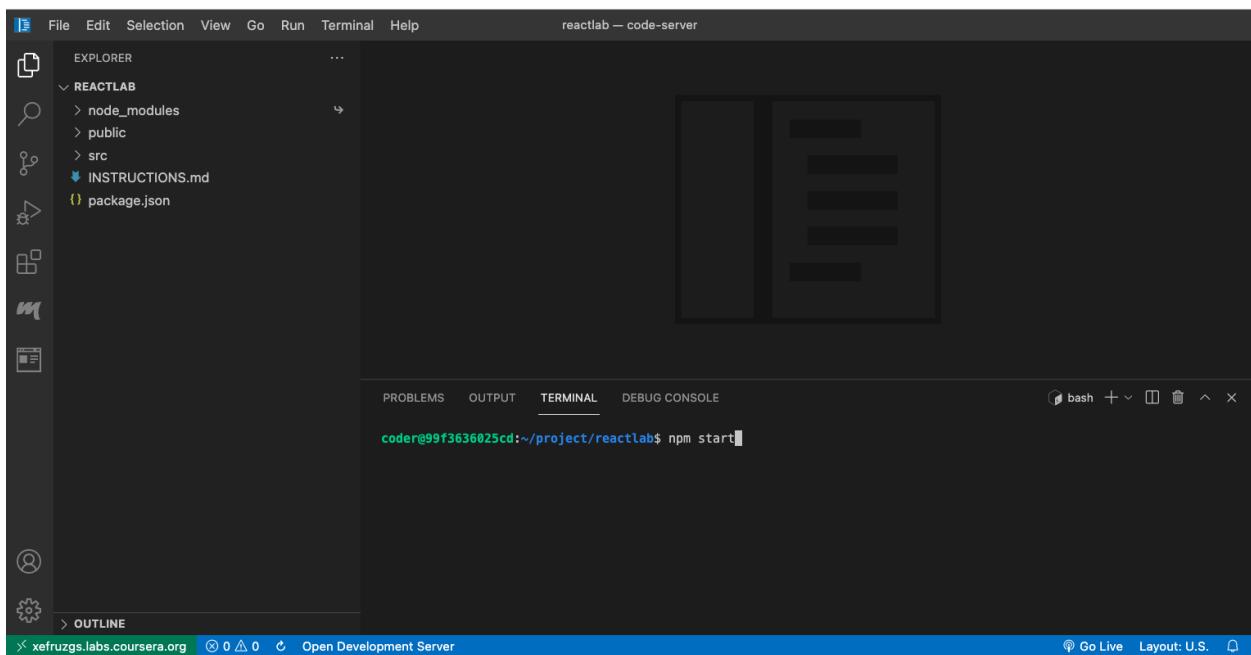
The labs for this course already have everything installed and setup so you can start working with React right away.



In order to run and view your React app you will need to open the VS Code built-in terminal, run **npm start**, and then click **Open Development server**.

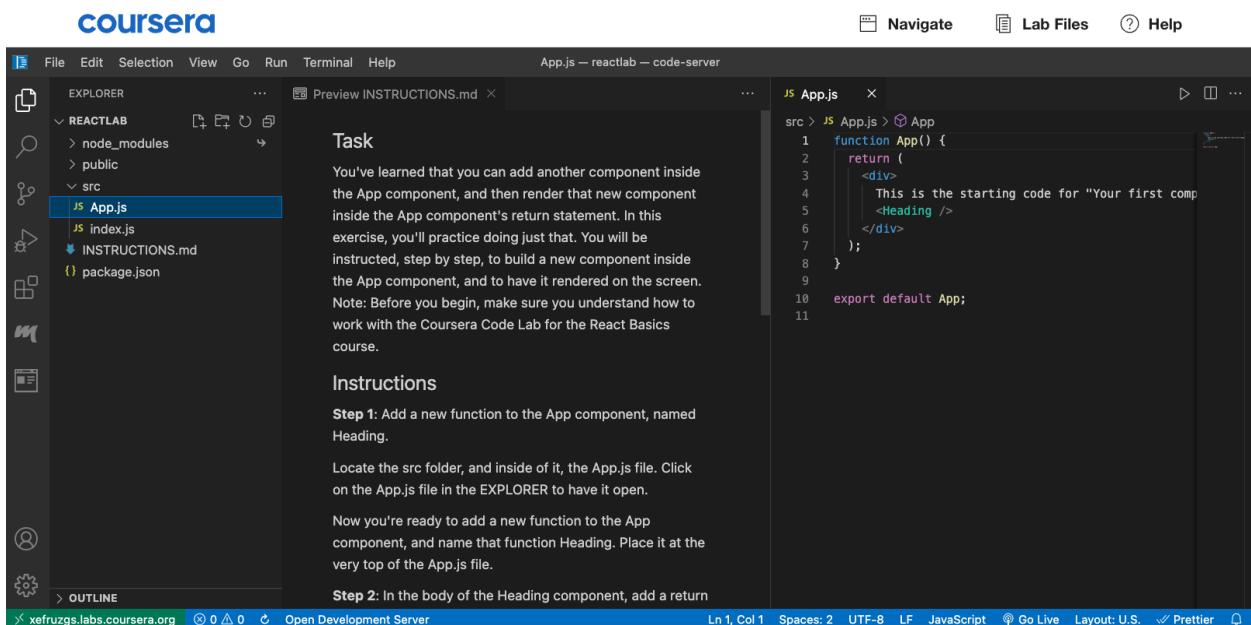
The **Open Development server** link can be found on the blue horizontal bar at the very bottom of the lab window, written in white letters on a blue background. Click this link.

That will open the app in the browser in a separate tab.



To view your code and instructions side-by-side, select the following in your VS Code toolbar:

1. View -> Editor Layout -> Two Columns
2. To view a file in Preview mode, right click on the file and open Preview (in the EXPLORER sidebar)
3. Select your code file in the code tree, which will open it up in a new VS Code tab.
4. You can drag any file over to the second column to view the contents in that column.
5. Great work! You can now see instructions and code at the same time.



Completed

Additional reading

Below you will find links to helpful additional readings.

- nodejs.org
- npmjs.com
- reactjs.org
- <https://create-react-app.dev/>
- [VS Code](#)

Completed

Setting up a React project in VS Code (Optional)

To complete the exercises in this course you have been provided with a dedicated lab environment set up specifically for you to apply the skills that you have learned. You can find out more about Working with Labs in this course by accessing the link below:

[Working with Labs in this course](#)

You can also use VS Code to practice these exercises on your local machine as an alternative option.

To follow along in this reading, you need to have Node.js and VS Code already installed on your computer. If you don't have this setup, please refer to the Programming with JavaScript course:

- [Setting up VS Code](#)
- [Installing Node and NPM](#)

In VS Code, you're ready to start a brand new React project.

You can do it using npm.

What is npm?

When Node.js is installed on a computer, **npm** comes bundled with it.

With **npm**, you can:

1. Author your own Node.js modules ("packages"), and publish them on the npm website so that other people can download and use them
2. Use other people's authored modules ("packages")

So, ultimately, npm is all about **code sharing** and **reuse**. You can **use other people's code** in your own projects, and you can also **publish your own Node.js modules** so that other people can use them.

An example npm module that can be useful for a new React developer is [create-react-app](#). While this npm module comes with its own website, you can also find some info on the [create-react-app project on GitHub](#).

Whenever you run the npm command to add other people's code, **that code, and all other Node modules that depend on it, get downloaded** to your machine.

However, although it's possible do to so, this is not really necessary, at least in the case of the [create-react-app](#) Node module.

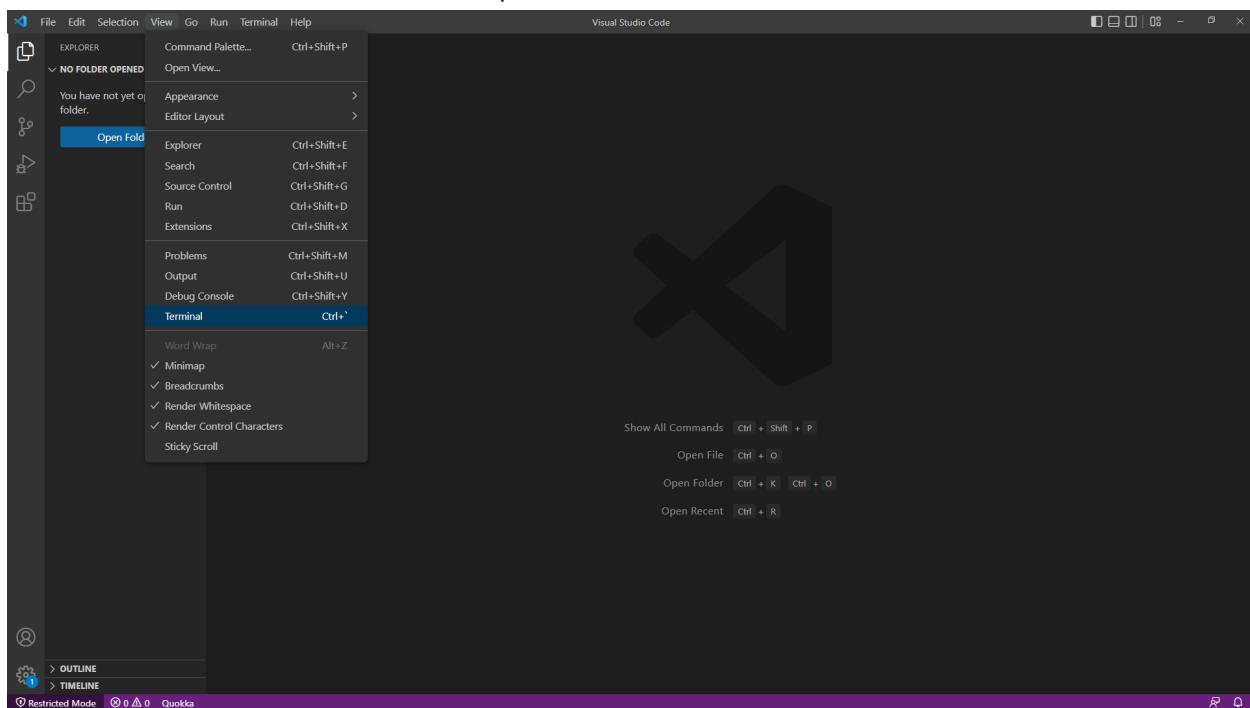
In other words, you can avoid installing the [create-react-app](#) package but still use it.

You can do that by running the following command: `npm init react-app example`, where "example" is the actual name of your app. You can use any name you'd like, but it's always good to have a name that is descriptive and short.

In the next section, you'll learn how to build a brand new app that you can name: `firstapp`.

Opening the built-in VS Code terminal and running *npm init react-app* command

In VS Code, click on *View*, *Terminal* to open the built-in terminal.



Now run the command to add a brand new React app to the machine:

1

```
npm init react-app firstapp
```

The installation and setup might take a few minutes.

Here's the output of executing the above command:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

Creating a new React app in /home/pc/Desktop/firstapp.

Installing packages. This might take a couple of minutes.

Installing react, react-dom, and react-scripts with cra-template...

added 1383 packages in 56s

190 packages are looking for funding

run `npm fund` for details

Initialized a git repository.

Installing template dependencies using npm...

npm WARN deprecated source-map-resolve@0.6.0:

See <https://github.com/lydell/source-map-resolve#deprecated>

added 39 packages in 6s

190 packages are looking for funding

run `npm fund` for details

Removing template package using npm...

```
removed 1 package, and audited 1422 packages in 3s
```

```
190 packages are looking for funding
```

```
    run `npm fund` for details
```

```
6 high severity vulnerabilities
```

```
To address all issues (including breaking changes), run:
```

```
npm audit fix --force
```

```
Run `npm audit` for details.
```

```
Created git commit.
```

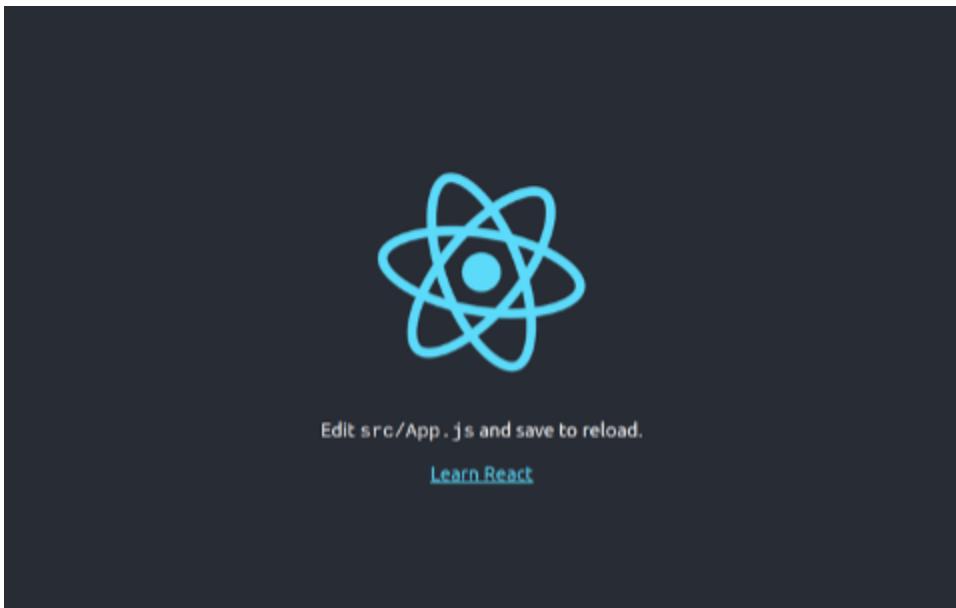
```
Success! Created firstapp at /home/pc/Desktop/firstapp
```

```
Inside that directory, you can run several commands:
```

If you follow the suggestions from the above output, you'll run: `cd firstapp`, and then `npm start`. This will end up with the following output in the built-in terminal:

*Compiled successfully! You can now view firstapp in the browser. Local:
<http://localhost:3000> On Your Network: <http://192.168.1.167:3000> Note that the development build
is not optimized. To create a production build, use npm run build. webpack compiled successfully*

Again, following the instructions, opening a browser with the address bar pointing to <http://localhost:3000>, will show the following page in your browser:



This means that you've successfully:

- Set up your local development environment
- Run the create-react-app npm package (without installing it!)
- Built a starter React app on your local machine
- Served that starter React app in your browser

After you've built your starting setup, in Module 2 you'll start working with the basic building blocks of React: components.

Completed

Transpiling JSX

By the end of this reading, you will have learned how a component is built.

Introduction

Components are a nice way to build websites in React because they allow you to build more modular apps. However, how do you build components using React, JSX, and JavaScript? You'll learn how this works in this lesson item.

A browser cannot understand JSX syntax.

This means that making a browser understand React code requires a lot of supporting technologies. An example of such a technology is a **transpiler**.

A **transpiler** takes a piece of code and transforms it into some other code.

To understand why this is done, here is an example of an ES6 variable declaration:

1

```
const PI = 3.14
```

This is perfectly valid ES6 syntax.

However, if you were using a very old computer, that computer will have an old browser. Perhaps that browser was built before ES6 came out in 2015.

This means that the JavaScript engine that is built into your old computer's browser is likely to be an ES5 JavaScript engine.

In ES5, the only way to declare a variable is the following:

1

```
var pi = 3.14
```

What this means is that for this old browser to understand the ES6 code, the only way to do it is by **transpiling** it.

If you feel like it, you can try transpiling ES6 to ES5 code yourself, using [the es6console website](#). Now, let's move the focus to another example of transpiling.

Let's say that you want to use a brand new, most modern ECMAScript syntax in an app. The only problem is that this new syntax is currently not supported by any browser; even an up-to-date browser.

However, by transpiling the new most-modern JavaScript syntax into something that modern browsers can understand, it is able to convert some code that the browser cannot comprehend, into code that it can comprehend, run, and produce a result from.

Likely the most popular site that shows off how this works is [Babel](#). As the heading of the website reads, "Babel is a JavaScript Compiler".

This finally brings you to the point of this discussion about transpiling JavaScript code.

What Babel does is this: it allows you to transpile JSX code (which cannot be understood by a browser) into plain JavaScript code (which can be understood by a browser).

This is where React and JSX come in.

For React code to be understood by a browser, you need to have a **transpiling step** in which the JSX code gets converted to plain JavaScript code that a modern browser can work with.

To demonstrate how this works, let's use the **Heading** component from the previous lesson.

Add the JSX code into [the online Babel repl](#). Repl stands for "read-eval-print loop" and it accepts code you write, evaluates it, and produces some result. In the specific case of [the online Babel repl](#), that result is some transpiled code. Here's a more detailed explanation.

If you've visited the above-linked URL, you'll find a web page that has two panels. On the left, there's source JSX code:

1

2

3

```
function Heading(props) {  
  
  return <h1>{props.title}</h1>  
  
}
```

... and on the right, there's the transpiled, plain JavaScript code:

1

2

3

4

5

```
"use strict";  
  
function Heading(props) {  
  
  return /*#__PURE__*/React.createElement("h1", null, props.title);  
  
}
```

If you now analyze the difference between the source JSX code and the transpiled, plain JavaScript code, dis-regarding the comment, here's the body of the Heading function:

1

2

```
React.createElement("h1", null, props.title);
```

So, here you have a React object, and this object has a `createElement()` method on it. The method is invoked with three arguments:

1. `"h1"`
2. `null`
3. `props.title`

The first argument is the DOM element to render - in this case, an `h1` element. The second property is any HTML attribute that should be added, and there's a null here - meaning, there should be an object with some data, but there isn't any data so instead of the object there's the null value. The third property is the contents of the inner HTML of the DOM element specified as the first argument - in this case, the contents of the inner HTML of the `h1` element.

Now let's use Babel again, and this time transpile the `render` syntax for the `Heading` component:

```
1 <Heading title="This is the heading text!"></Heading>
```

Again using [the Babel repl](#), and as can be confirmed in [the link](#), the output of the transpilation is the following code:

```
1 "use strict";  
2  
3 /*#__PURE__*/  
4 React.createElement(Heading, {  
5   title: "This is the heading text!"  
6 });
```

Again, you have the `React.createElement()` method call, and this time, the first item to render is `Heading`, and then you have an object as the second argument (instead of a null that you had in the previous transpilation example).

This brings me to an interesting question: What is the minimum code that a component must have to be able to show something on the screen when rendered?

You can see the answer below:

```
1  
2  
3  
4  
  
function Example() {  
  
  return <div>An element</div>  
  
}  
  
export default Example
```

Completed

Solution: Your first component

Here is the completed solution code for the App.js file:

```
1  
2  
3  
4
```

```
5
6
7
8
9
10
11
12
13
14
15
16

function Heading() {
    return (
        <h1>This is an h1 heading.</h1>
    )
}

function App() {
    return (
        <div className="App">
```

```
This is the starting code for "Your first component" ungraded lab

<Heading />

</div>

);

}

export default App;
```

Here is the output from the solution code for the App.js file:



A screenshot of a web browser window. The address bar shows the URL: exbsbtqo.labs.coursera.org/serve/. The main content area displays the text: "This is the starting code for "Your first component" ungraded lab".

This is an h1 heading.

Step 1: In the starting code, you already had a JSX element named `<Heading />`, being rendered from the App component, since it is a part of the App component's return statement.

1

2

3

4

5

6

7

8

9

10

```
function App() {  
  return (  
    <div className="App">  
      This is the starting code for "Your first component" ungraded lab  
      <Heading />  
    </div>  
  );  
  
}  
  
export default App;
```

Then, you added a new function to the App component, and named that function `Heading ()`. You placed it at the very top of the `App.js` file.

1

2

```
function Heading() {  
}  
  
1  
2
```

Step 2: Next, in the body of the `Heading` component, you added a return statement and spread it over several lines by following it up with an opening and a closing parenthesis.

1

2

3

4

```
function Heading() {  
}  
  
1  
2  
3  
4
```

```
    return (  
        )  
    }  
}
```

Step 3: Then, inside the parentheses, you added the following code: <h1>This is an h1 heading</h1>

1
2
3
4
5

```
function Heading() {  
    return (  
        <h1>This is an h1 heading.</h1>  
    )  
}
```

Step 4: Finally, you saved your changes and viewed the app in the browser.
Completed

Customizing the project

So far, you've learned about React components, but now you will focus on learning how to customize the project. You will learn about the software development approach, detailing the creation of separate associated files, the requirements gathering and the subsequent folder structure to be created.

Building a Layout

Imagine that you've been given the task of building a somewhat more complex website layout using React.

At this point, you still don't know too much about how React works, but even with your limited knowledge, you can still build some relatively interesting designs.

Currently, you need to build a simple typography-focused layout for a coding blog.

This means that you will not have to use images, which simplifies your task significantly.

The layout you're supposed to build will consist of the following sections:

- Main navigation
- Promo (main advertisement)
- A list of newest posts' previews (intros)
- The footer

Organizing Your Code

Keeping in mind the above structure, how would you organize your code?

This is where [React docs](#) can help. They suggest two approaches:

1. Grouping by features
1. Grouping by file type

They also advise not to nest folders too deep, and to keep things simple and not overthink it.

They even say that if you're just starting out, you shouldn't spend more than five minutes setting up a project.

Taking this advice into account, you might say that for a small project like this, you could keep it as simple as just adding a **components** folder and moving all your components into it. This is exactly what you'll do next.

Building The App

Since this app's focus is on customization, let's name the app **customizing-example**.

What follows is the command to run **in a suitable folder on your own computer**. By "a suitable folder", I mean: "a folder where you feel comfortable installing a boilerplate React application". This also includes that the folder you chose will need to be accessible for your user on your OS (Operating System).

1

```
npm init react-app customizing-example
```

This will produce a brand-new starter app with a familiar structure.

Inspecting the **src** folder of the starter app, it looks like this:

1

2

3

4

5

6

7

8

src/

 App.js

 App.test.js

 index.css

 index.js

 logo.svg

 reportWebVitals.js

 setupTests.js

Then simply add a components folder to it, like this:

1

2

3

4

5

6

7

8

9

```
src/
  components/
    App.js
    App.test.js
  index.css
  index.js
  logo.svg
  reportWebVitals.js
  setupTests.js
```

Since the components folder is currently empty, you can add a component for each of the sections of the typography-focused blog. Here's the structural update:

1

2

3

4

5

6

7

8

9

10
11
12
13
14
15

src/
components/
 Nav.js
 Promo.js
 Intro1.js
 Intro2.js
 Intro3.js
 Footer.js
 App.js
 App.test.js
index.css
index.js
logo.svg
reportWebVitals.js
setupTests.js

At this point, there's no need to complicate things. You have the **Nav** component, the **Promo** component, the **Intro1**, **Intro2**, and the **Intro3** component. Finally, there's also a **Footer.js** component.

This means you've fully planned the app, based on some best practices as suggested by the official React docs website, and based on the level of complexity of the project itself. Since this project is relatively simple, this structure feels right.

In this reading, you'll just build all the components inside the components folder, and then, in the upcoming lesson items, import them into the **App.js** file.

Building Components

For now, let's just build those components. After you've added the **components** folder, you've also added all the functional component files. Since they are all currently empty, you can start adding them, one by one.

Heres' the contents of the **Nav.js** file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

```
function Nav() {  
  
  return (  
  
    <nav className="main-nav">  
  
      <ul>  
  
        <li>Home</li>  
  
        <li>Articles</li>  
  
        <li>About</li>  
  
        <li>Contact</li>  
  
      </ul>  
  
    </nav>  
  
  );  
  
}  
  
export default Nav;
```

Next, you can focus on the **Promo.js** file:

1

2

3

4

5

6

```
7
8
9
10
11
12
13
14

function Promo() {
  return (
    <div className="promo-section">
      <div>
        <h1>Don't miss this deal!</h1>
      </div>
      <div>
        <h2>Subscribe to my newsletter and get all the shop items at 50% off!</h2>
      </div>
    </div>
  );
}
```

```
export default Promo;
```

Once you've finished the promo section, you can focus on the Intro components. Here's `Intro1.js`:

```
1
2
3
4
5
6
7
8
9
10
11
12
13

function Intro1() {
  return (
    <div className="blog-post-intro">
      <h2>I've become a React developer!</h2>
      <div>
```

```
<p>I've completed the React Basics course and I'm happy to
announce that I'm now a Junior React Developer!</p>

<p className="link">Read more...</p>

</div>

</div>

);

};

export default Intro;
```

Here's the code for the **Intro2.js** component:

1
2
3
4
5
6
7
8
9
10
11
12

```
function Intro2() {  
  
  return (  
  
    <div className="blog-post-intro">  
  
      <h2>Why I love front-end web development</h2>  
  
      <div>  
  
        <p>In this blog post, I'll list 10 reasons why I love to  
work as a front-end developer.</p>  
  
        <p className="link">Read more...</p>  
  
      </div>  
  
    </div>  
  
  );  
  
}  
  
export default Intro2;
```

You can finish the previews for my blog posts with the code for **Intro3.js** component:

1

2

3

4

5

6

```
7
8
9
10
11
12
13

function Intro3() {
  return (
    <div className="blog-post-intro">
      <h2>What's the best way to style your React apps?</h2>
      <div>
        <p>There are so many options to choose from. Here's a
        high-level overview of the popular ones.</p>
        <p className="link">Read more...</p>
      </div>
    </div>
  );
}

export default Intro3;
```

There's just one more thing left to code, the **Footer** component, so here it is:

```
1
2
3
4
5
6
7
8
9

function Footer() {
  return (
    <div className="copyright">
      <p>Made with love by Myself</p>
    </div>
  );
}

export default Footer;
```

Now that you have completed all the components for the app, here are a few more interesting things about the syntax.

These are:

- The use of the `className` attribute in JSX
- The use of separate components for repetitive code

- Where are all the props?
- Why was I not using the `<a>` element for empty links?

Discussing the Syntax

Now let's briefly discuss the four bullet points above.

Why use the `className` attribute in the JSX syntax?

Well, with JSX, it looks like HTML so much that it's easy to forget that it's actually JavaScript code - not HTML.

While regular HTML does indeed have a `class` attribute, which is used to list one or more CSS classes to be used on a given HTML element, this cannot really work in JSX. The reason is that JSX is a special kind of JavaScript syntax, and the word `class` is a reserved keyword in JSX. That's why the React team had to make a compromise and so `className` is used in JSX to list one or more CSS classes to be used on a given element or component.

But why use `Intro1.js`, `Intro2.js`, and `Intro3.js`? Isn't one of the tenets of coding the DRY approach - that is, the "Don't repeat yourself" approach?

Indeed, it is. However, there are still a few concepts to discuss before you learn how to re-use a single component with variations in its content. This has to do with data in components, but don't worry, we'll be getting to that later.

The third question is about the `props` object. It has been mentioned before, but so far it hasn't been used. It hasn't been used in this example either.

The answer to this question has to do with the next lesson, titled ***Component Use and Styling***. In this lesson, you'll see in practice how you can make components work better, with the help of `props`.

The final question is about not using the `<a>` element for empty links in my app.

The answer here depends on whether those links are "internal" - inside an app, or "external", meaning, leading to some external link, such as <https://www.coursera.org>. If the links are internal to the app - as they are envisioned here - using the `<a>` tag is simply not the React way of doing things. You'll learn why that is the case when discussing the use of React Router.

Conclusion

Having finished this reading, you have now learned about the software development approach, detailing the creation of separate associated files, the requirements gathering, and the subsequent folder structure to be created.

Completed

Solution: Creating and importing components

Here are the contents of the `Heading.js` file:

```
2
3
4
5
6
7

function Heading() {
  return (
    <h1>This is an h1 heading</h1>
  )
}

export default Heading;
```

Here are the contents of the App.js file:

```
1
2
3
4
5
6
7
```

8

9

10

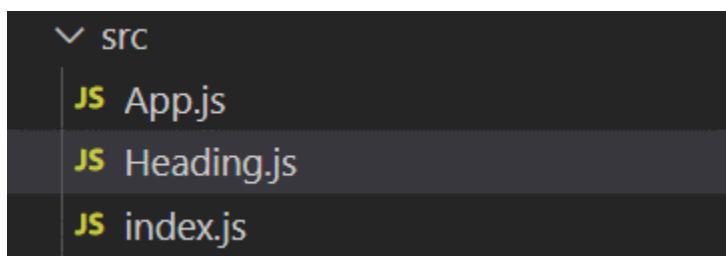
11

```
import Heading from './Heading';

function App() {
  return (
    <div className="App">
      <Heading />
    </div>
  );
}

export default App;
```

Here is a screenshot of the src folder:



Here is the output from the solution code for the App.js file:

This is an h1 heading

Step 1: You moved the `Heading` function from `App` to a separate component file, named “`Heading.js`”.

```
1  
2  
3  
4  
5  
6  
7  
  
function Heading() {  
  return (  
    <h1>This is an h1 heading</h1>  
  )  
}  
  
export default Heading;
```

Step 2: Next, you imported the `Heading` component into the `App` component.

```
1  
  
import Heading from "./Heading";
```

Step 3: Finally, you removed the sentence that reads: *This is the starting code for “Your first component” ungraded lab* - so that only the `Heading` JSX element remains in the return statement of the App component.

```
1
2
3
4
5
6
7
8
9
10
11

import Heading from "./Heading";

function App() {
  return (
    <div className="App">
      <Heading />
    </div>
  );
}
```

```
export default App;
```

Completed

Additional resources for React components and where they live

Below you will find links to helpful additional resources.

- [Basic Concepts of Flexbox](#)
- [Importing a Component](#)
- [Babeljs.io](#)
- [NPM docs: package.json](#)
- [git docs: gitignore](#)
- [NPM docs: node modules folder](#)
- [webpack docs: DevServer](#)
- [webpack/webpack-dev-server on GitHub](#)
- [Visual Studio Code keyboard shortcuts \(Windows\)](#)
- [Visual Studio Code keyboard shortcuts \(macOS\)](#)

Completed

Dissecting props

Recall that much like parameters in a JavaScript function which allow you to pass in values as arguments, React uses properties, or **props**, to pass data between components. But how exactly do they work?

In this reading, you'll use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.

Remember first that JSX code in React is just syntactic sugar - meaning, a nicer way to write some hard-to-read code.

For the browser to understand this syntactic sugar, you need to transpile JSX down to plain JavaScript code. You have a resource online, at the URL of [babeljs.io](#), which allows you to inspect

the results of this transpiling. Once you visit the website, make sure to navigate to the *Try it out* link in the main navigation.

For example, let's say you have a component that returns a piece of JSX:

```
1  
2  
3  
  
function App() {  
  
  return <h1>Hello there</h1>  
  
}  
  
... if you used the Babel transpiler to transpile this JSX syntactic sugar code down to plain JavaScript code, you'd get back some unusual code:
```

```
1  
2  
3  
4  
  
"use strict";  
  
function App() {  
  
  return /*#__PURE__*/React.createElement("h1", null, "Hello there");  
  
}
```

You just want to focus on the `React.createElement("h1", null, "Hello there")`; part. You can ignore the rest.

This means that the `createElement` function receives three arguments:

1. The wrapping element to render.
2. A null value (which is there to show an absence of an expected JavaScript object value).
3. The inner content that will go inside the wrapping element.

Interestingly, the inner content that will go inside the wrapping element can also be a call to the `createElement` function.

For example, let's say you have a slightly more complex JSX element structure:

```
1  
2  
3  
4  
5  
6  
7  
  
function App() {  
  return (  
    <div>  
      <h1>Hello there</h1>  
    </div>  
  )  
}  
}
```

... the transpiled return statement in plain JavaScript again returns two `createElement` functions:

```
1  
2  
3  
4  
  
"use strict";  
  
function App() {
```

```
    return /*#__PURE__*/React.createElement("div", null,
  /*#__PURE__*/React.createElement("h1", null, "Hello there"));
}

If you format this output, remove the "use strict" line, and remove the __PURE__ comments, you
get a more readable output:
```

```
1
2
3
4
5
6
7

function App() {
  return React.createElement(
    "div",
    null,
    React.createElement("h1", null, "Hello there")
  );
}
```

So now the third argument of the outer-most `React.createElement` call is another `React.createElement` call.

This is how you can nest as many elements as you want.

This means that a nested JSX structure is just a bunch of nested `React.createElement` calls, passed in to other `React.createElement` calls as their third argument.

The second – null – argument

The second argument of `null` can – in this case – be replaced with an empty object. In that case, your code would contain a pair of curly braces instead of the word `null`:

```
1
2
3
4
5
6
7
8
9

"use strict";

function App() {
  return React.createElement(
    "div",
    {},
    React.createElement("h1", {}, "Hello there")
  );
}
```

This object is referred to as the *props* object. It is the main mechanism of sending data from a parent component to a child component in React.

The way this works is described in React docs using the following code:

1

2

3

4

5

```
React.createElement(  
  type,  
  [props],  
  [...children]  
)
```

The third argument (...children)

This is the inner content that will go inside the wrapping element. It's what makes it possible to nest elements inside other elements, mimicking the way that HTML works.

In this reading you've learned how to use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.

Completed

Solution: Passing props

Here is the completed App.js file:

1

2

3

4

```
5

6

7

8

9

10

11

12

import Heading from "./Heading";

function App() {
  return (
    <div className="App">
      <Heading firstName="Bob" />
      <Heading firstName="Any name other than Bob" />
    </div>
  );
}

export default App;
```

And here is the completed `Heading.js` file:

```
1
2
3
4
5
6
7
8

function Heading(props) {
  return (
    <h1>Hello, {props.firstName}</h1>
  )
}

export default Heading;
```

Here is the output from the solution code for the App.js file:

Hello, Bob

Hello, Any name other than Bob

Step 1: First, you passed the props object as a parameter to the `Heading` component in the 'Heading.js' file.

```
1
2
3
4
5
6
7
8

function Heading(props) {
  return (
    <h1>Hello, </h1>
  )
}

export default Heading;
```

Step 2: Next, inside the `Heading` component's body, you located the `return` statement, and added a JSX expression that accesses the `firstName` property of the `props` object, inside the `return` statement's `h1`, after `Hello`.

```
1
2
3
4
5
6
7
8

function Heading(props) {
  return (
    <h1>Hello, {props.firstName}</h1>
  )
}

export default Heading;
```

Step 3: Then, inside the `App` component's `return` statement, you located the `<Heading />` JSX element, and added the attribute of `firstName` and give it the value of `Bob`.

1

2

```
3
4
5
6
7
8
9
10
11
12

import Heading from "./Heading";

function App() {
  return (
    <div className="App">
      <Heading firstName="Bob" />
    </div>
  );
}

export default App;
```

Step 4: You saved all your changes and ran the app to preview the updates in the browser, and confirm that the page shows an `h1` element with the text that reads "Hello, Bob".

Step 5: Then, you changed `firstName` to any name other than 'Bob' and see how the page updates with the new name.

```
1
2
3
4
5
6
7
8
9
10
11
12

import Heading from './Heading';

function App() {
  return (
    <div className="App">
      <Heading firstName="Any name other than Bob" />
    </div>
  );
}

export default App;
```

```
</div>

);

}

export default App;
```

Step 6: Finally, you added a second `<Heading />` after the first one. Again, adding the `firstName` attribute and choosing another name for this value.

```
1
2
3
4
5
6
7
8
9
10
11
12
13

import Heading from "./Heading";
```

```
function App() {  
  
  return (  
  
    <div className="App">  
  
      <Heading firstName="Any name other than Bob" />  
  
      <Heading firstName="Jack" />  
  
    </div>  
  
) ;  
  
}  
  
export default App;
```

Completed

Props and children

Previously, you learned that you could pass props to and within a component. But there is also a special prop known as `props.children`, which is automatically passed to every component. In this reading, you'll learn about `props.children` and what its purpose is.

To understand the concept of `props.children`, consider the following real-life situation: you have a couple of apples, and you have a couple of pears. You'd like to carry the apples some distance, so obviously, you'll use a bag.

It's not a "bag for apples". It's not a "bag for pairs". It's just a bag. Nothing about this bag makes it such that it needs to be referred to as a bag in which you'd only and always carry apples, nor a bag in which you'd only and always carry pears.

In a way, the bag "doesn't care" if it is used to carry apples or pears. Nothing about the bag changes. There are no changes in the bag's material, size, shape, or color - because it can handle apples or pears being carried inside of it, without issues.

Now, consider the following component:

```
1
2
3
4
5
6
7
8
9
10
11
12
13

function Apples(props) {
  return (
    <div className="promo-section">
      <div>
        <h2>These apples are: {props.color}</h2>
      </div>
    </div>
  )
}
```

```

        <div>

            <h3>There are {props.number} apples.</h3>

        </div>

    </div>

)

}

export default Apples

```

There is also a **Pears** component:

```

1

2

3

4

5

function Pears(props) {

    return (

        <h2>I don't like pears, but my friend, {props.friend}, does</h2>

    )

}

```

Now, the question is this: Let's say you want to have a **Bag** component, which can be used to "carry" **Apples** or **Pears**. How would you do that?
This is where **props.children** comes in.
You can define a **Bag** component as follows:

```

1

2

```

```
3
4
5
6
7
8
9
10
11
12
13
14

function Bag(props) {
  const bag = {
    padding: "20px",
    border: "1px solid gray",
    background: "#fff",
    margin: "20px 0"
  }
  return (
    <div style={bag}>
```

```
        {props.children}

    </div>

)

}

export default Bag
```

So, what this does in the `Bag` component is: it adds a wrapping `div` with a specific styling, and then gives it `props.children` as its content.

But what is this `props.children`?

Consider a very simple example:

```
<Example>
```

```
    Hello there
```

```
</Example>
```

The `Hello there` text is a child of the Example JSX element. The Example JSX Element above is an "invocation" of the `Example.js` file, which, in modern React, is usually a function component.

Now, did you know that this `Hello there` piece of text can be passed as a **named prop** when rendering the `Example` component?

Here's how that would look like:

```
<Example children="Hello there" />
```

Ok, so, there are two ways to do it. But this is just the beginning.

What if you, say, wanted to surround the `Hello there` text in an `h3` HTML element? Obviously, in JSX, that is easily achievable:

```
<Example children={<h3>Hello there</h3>} />
```

What if the `<h3>Hello there</h3>` was a separate component, for example, named `Hello`? In that case, you'd have to update the code like this:

1

```
<Example children={<Hello />} />
```

You could even make the `Hello` component more dynamic, by giving it its own prop:

1

```
<Example children={<Hello message="Hello there" />} />
```

So, given the **Bag**, **Apples**, and **Pears** examples from the beginning of this reading, armed with this new knowledge, how can you make it work?

Here's how you'd render the `Bag` component with the `Apples` component as its `props.children`:

1

```
<Bag children={<Apples color="yellow" number="5" />} />
```

And here's how you'd render the `Bag` component, wrapping the `Pears` component:

1

```
<Bag children={<Pears friend="Peter" />} />
```

While the above syntax might look a bit weird, it's important to understand what is happening "under the hood".

Effectively, the above syntax is the same as the two examples below.

1

2

3

4

5

6

7

```
<Bag>
```

```
<Apples color="yellow" number="5" />  
</Bag>
```

```
<Bag>  
  
<Pears friend="Peter" />  
  
</Bag>
```

You can even have multiple levels of nested JSX elements, or a single JSX element having multiple children, such as, for example:

```
1  
2  
3  
4  
5  
6
```

```
<Trunk>  
  
<Bag>  
  
<Apples color="yellow" number="5" />  
  
<Pears friend="Peter" />  
  
</Bag>  
  
</Trunk>
```

So, in the above structure, there's a `Trunk` JSX element, inside of which is a single `Bag` JSX element, holding an `Apples` and a `Pears` JSX element.

Before the end of this reading, consider this JSX element again:

1

2

3

```
<Bag>  
  <Apples color="yellow" number="5" />  
</Bag>
```

What is **Apples** to **Bag** in the above code?

In the above code, **Apples** is a prop of the **Bag** component. To explain further, the Bag component can wrap the Apples component, or any other component, because I used the **{props.children} syntax in the Bag component function declaration**. In other words, just like in the real world, when you take a bag to a grocery store, you can “wrap” a wide variety of groceries inside the bag, you can do the same thing in React: wrap a wide variety of components inside the **Bag** component, using the children prop to achieve this.

It's crucial to understand this when working with React.

Before the end of this reading, there's another important concept that you need to be aware of: *finding the right amount of modularization*. What does this mean? Imagine, for example, that you had a number of small bags, and that each bag could only carry a single apple or pear. You'd end up having to wrap each "apple" inside a "bag". That doesn't make much sense. You can think about components making your layouts modular in a similar way. You don't want to have an entire layout contained in a single component, because that would be very difficult to work with. On the flip side, if you made each HTML element in your layout a separate component, that would make it very hard to work with, although such layout would be modular. So it's all about moderation. You need to organize your layouts by splitting them into meaningful areas of the page, and then code those meaningful areas as separate components. that would constitute the right amount of modularity. To reinforce this point, It might help to think of it in terms of how a person would describe a website: there's a menu, a footer, the shopping cart, etc.

In conclusion, when you see a JSX element wrapping some other JSX element, you can easily understand that it's all just **props.children** in the background.

Completed

Styling JSX elements

You've observed that JSX is incredibly versatile, and can accept a combination of JavaScript, HTML and CSS. In this reading, you'll learn some approaches for styling JSX elements and doing so in a way that achieves both a functional and visual aspect within an app.

There are various ways to style JSX elements.

Probably the simplest way to do this is using the `link` HTML element in the head of the `index.html` file in which your React app will mount.

The `href` attribute loads some CSS styles, probably with some CSS classes, and then, inside the function component's declarations, you can access those CSS classes using the `className` attribute.

```
1
2
3
4
5
6
7
8
9
10
11
12

function Promo(props) {
  return (
    <div className="promo-section">
      <div>
        <h1>{props.heading}</h1>
```

```
</div>

<div>

  <h2>{props.promoSubHeading}</h2>

</div>

</div>

) ;

}
```

In CSS:

```
1

2

3

4

.promo-section {

  font-weight: bold;

  line-height: 20px;

}
```

Another way to add CSS styles to components is using inline styles.

The syntax of inline styles in JSX is a bit custom.

Consider a starting `Promo` component, containing code that you encountered earlier:

```
1

2

3

4
```

```
5
6
7
8
9
10
11
12
13
14

function Promo(props) {
  return (
    <div className="promo-section">
      <div>
        <h1>{props.heading}</h1>
      </div>
      <div>
        <h2>{props.promoSubHeading}</h2>
      </div>
    </div>
  );
}
```

```
}
```

```
export default Promo;
```

Now you can add some inline styles to it:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

function Promo(props) {
```

```

    return (
      <div className="promo-section">
        <div>
          <h1 style={{color:"tomato", fontSize:"40px",
          fontWeight:"bold" }}>
            {props.heading}
          </h1>
        </div>
        <div>
          <h2>{props.promoSubHeading}</h2>
        </div>
      </div>
    );
  }
}

export default Promo;

```

You can start updating the `Promo` component by adding the JavaScript expression syntax:

[1](#)

```
<h1 style={ }>
```

As explained previously, this means that whatever code you add inside these opening and closing curly braces is to be parsed as regular JavaScript. Now let's add a **style object literal** inside of these curly braces:

[1](#)

```
<h1 style={{color:"tomato",fontSize:"40px" }}>
```

You can then re-write this object literal:

```
1
2
3
4

{
  color: "tomato",
  fontSize: "40px"

}
```

So, there's nothing special about this object, except for the fact that you've inlined it and placed it inside a pair of curly braces. Additionally, since it's just JavaScript, those CSS properties that would be hyphenated in plain CSS, such as, for example, `font-size:40px`, become camelCased, and the value is a string, making it look like this: `fontSize:"40px"`.

Besides inlining a *style object literal*, you can also save it in a variable, and then use that variable instead of passing an object literal.

That gives you an updated `Promo` component, with the `styles` object saved as a JavaScript variable:

```
1
2
3
4
5
6
7
8
9

const styles = {
  color: "tomato",
  fontSize: "40px"
}

function Promo() {
  return (
    <div>
      <h1>Promo</h1>
    </div>
  )
}
```

```
10
11
12
13
14
15
16
17
18
19
20

function Promo(props) {
  const styles = {
    color: "tomato",
    fontSize: "40px"
  }

  return (
    <div className="promo-section">
      <div>
```

```
<h1 style={styles}>  
  {props.heading}  
</h1>  
</div>  
  
<div>  
  <h2>{props.promoSubHeading}</h2>  
</div>  
</div>  
) ;  
}
```

Using this approach makes your components more self-contained, because they come with their own styles built-in, but it also makes them a bit harder to maintain.

Completed

JSX syntax and the arrow function

Components as Function Expressions

Up to this point, you've likely only observed ES5 function declarations used to define components in React. However, this is not the only way to do it.

In this reading, you learn about some alternative approaches, specifically by using function expressions and arrow functions.

Function Expressions

Let's start with a function declaration used as a component in React:

1

2

3

```
4  
5  
6  
7  
  
function Nav(props) {  
  
  return (  
  
    <ul>  
  
      <li>{props.first}</li>  
  
    </ul>  
  
  )  
  
}  
  
This component's code returns a list item containing the value of the 'first' prop.  
Now, let's change this function declaration to a function expression:  
1  
2  
3  
4  
5  
6  
7
```

```
const Nav = function(props) {  
  
  return (  

```

```
<ul>

  <li>{props.first}</li>

</ul>

)
```

The component is, for the most part, the same. The only thing that's changed is that you're now using an anonymous (nameless) function, and assigning this anonymous function declaration to a variable declared using the `const` keyword, and the name `Nav`. The rest of the code is identical. Changing a component from a function declaration to a function expression doesn't change its behavior, or how you write the code to render the `Nav` component. It's still the same:

1

```
<Nav first="Home" />
```

You can also take this concept a step further, using arrow functions.

Components as Arrow Functions

Arrow functions are a core feature of the ES6 version of JavaScript. One of the main benefits of using arrow functions is its shorter syntax. Consider the `Nav` function expression written as an arrow function:

1

2

3

4

5

6

7

```
const Nav = (props) => {

  return (
    <ul>
```

```
<ul>

  <li>{props.first}</li>

</ul>

)
```

So, the way to think about this is the following:

- The arrow itself can be thought of as the replacement for the `function` keyword.
- The parameters that this arrow function accepts are listed before the arrow itself.

To reiterate, take the smallest possible **anonymous ES5 function**:

[1](#)

```
const example = function() {}
```

And then observe how this is written as an arrow function:

[1](#)

```
const example = () => {}
```

Another important rule regarding arrow functions is that using the parentheses is optional if there's a single parameter that a function accepts.

In other words, another correct way to write the previous Nav arrow function component would be to drop the parentheses around 'props':

[1](#)

[2](#)

[3](#)

[4](#)

[5](#)

[6](#)

[7](#)

```
const Nav = props => {
```

```
return (
  <ul>
    <li>{props.first}</li>
  </ul>
)

}
```

In all other cases, when you write arrow functions, **for any number of parameters other than a single parameter, using parentheses around parameters is compulsory**.

For example, if your `Nav` component wasn't accepting any parameters, you'd code it with empty parentheses:

```
1
2
3
4
5
6
7
```

```
const Nav = () => {
  return (
    <ul>
      <li>Home</li>
    </ul>
  )
}
```

```
}
```

Another interesting thing about arrow functions is the **implicit return**. However, it only works if it's on the same line of code as the arrow itself. In other words, the implicit return works if your entire component is a single line of code.

To demonstrate how this works, let's re-write the `Nav` component as a one-liner:

1

```
const Nav = () => <ul><li>Home</li></ul>
```

Note that with the implicit return, you don't even have to use the curly braces that are compulsory function body delimiters in all other cases.

Using Arrow Functions in Other Situations

In React, just like in plain JavaScript, arrow functions can be used in many different situations. One such situation is using it with, for example, the `forEach()` built-in array method.

For example:

1

```
[10, 20, 30].forEach(item => item * 10)
```

The output of the above vanilla JavaScript line of code would be three number values:

100 200 300

As a side-note, the term "vanilla JavaScript" is often used to describe the plain, regular JavaScript language syntax, without any framework-specific or library-specific code. For example, React is a library, so in this context, saying that a piece of code is "vanilla JavaScript" means that it doesn't need any special library to run. It can run in "plain" JavaScript without any additional dependencies. You could also write this code in ES5 syntax:

1

2

3

4

```
[10, 20, 30].forEach(function(item) {  
  return item * 10  
})
```

Regardless of how you write it, the `forEach()` method can be run on an array. The `forEach()` method accepts a single parameter: **an anonymous function**. If you write this anonymous function in ES5 syntax, then it would contain a return statement:

1
2
3

```
function(item) {  
  
    return item * 10  
  
}
```

If you write it as an ES6 function instead, it can be simplified as one line:

1

```
item => item * 10
```

Both these functions perform the exact same task. Only the syntax is different. The ES6 function is a lot shorter because:

- The arrow function has a single parameter, so you do not need to add parentheses around the item parameter (to the left of the arrow)
- Since the arrow function fits on one line of code, you don't need to use curly braces around the function body, or the return keyword; it's implicit

Arrow functions are used extensively in JSX in React, and getting used to their syntax and being able to "mentally parse" it as you read it is an important skill to have and helps you get better at writing React apps.

Now that you have completed this reading, you've learned about some alternative approaches, specifically by using function expressions and arrow functions.

Completed

Ternary operators and functions in JSX

So you've explored several ways to define components in React; this includes function declarations, function expressions and arrow functions.

As you continue with building your knowledge of React syntax, you'll learn to make more use of JSX and embedded JSX expressions.

In this reading, you will become familiar with how to use ternary expressions to achieve a random return, as well as how to invoke functions inside of JSX expressions.

A different way of writing an if...else conditional

You are likely familiar with the structure of an if...else conditional. Here is a quick refresher:

```
1  
2  
3  
4  
5  
6  
7  
  
let name = 'Bob';  
  
if (name == 'Bob') {  
  
    console.log('Hello, Bob');  
  
} else {  
  
    console.log('Hello, Friend');  
  
};
```

The above code works as follows:

1. First, I declare a `name` variable and set it to a string of "Bob".
2. Next, I use the `if` statement to check if the value of the `name` variable is "Bob". If it is, I want to `console.log` the word "Bob".
3. Otherwise, if the `name` variable's value is not "Bob", the `else` block will execute and output the words "Hello, Friend" in the console.

Above, I gave you an example of using an `if...else` conditional. Did you know that there is another, different way, to effectively do the same thing? It's known as the **ternary operator**. A ternary operator in JavaScript uses two distinct characters: the first one is **the question mark**, that is,

the ? character. To the left of the ? character, you put a *condition that you'd like to check for*. Just like I did in the above `if...else` statement, the condition I'm checking is `name == 'Bob'`. In other words, I'm asking the JavaScript engine to look at the value that's stored inside the `name` variable, and to verify if that value is the same as '`Bob`'. If it is, then the JavaScript engine will return the boolean value of `true`. If the value of the `name` variable is something different from '`Bob`', the value that the JavaScript engine returns will be the boolean value of `false`.

Here is the code that reflects the explanation in the previous paragraph:

1

```
name == 'Bob' ?
```

Note that the above code is incomplete. I have the condition that I'm checking (the `name == 'Bob'` part). I also have the ? character, that is, the first of the two characters needed to construct a syntactically valid ternary operator. However, I still need the second character, which is the colon, that is the : character. This character is placed after the question mark character. I can now expand my code to include this as well:

1

```
name == 'Bob' ? :
```

This brings me a step closer to completing my ternary operator. Although I've added the characters needed to construct the ternary operator, I still need to add the return values. In other words, if `name == 'Bob'` evaluates to true, I want to return the words, "Yes, it is Bob!". Otherwise, I want to return the words "I don't know this person".

1

```
name == Bob ? "Yes, it is Bob" : "I don't know this person";
```

This, in essence, is how the ternary operator works. It's just some shorthand syntax that I can use as a replacement for the `if` statement. To prove that this is really the case, here's my starting `if...else` example, written as a ternary operator:

1

2

3

```
let name = 'Bob';

name == 'Bob' ? console.log('Hello, Bob') : console.log('Hello, Friend');
```

Using ternary expressions in JSX

Let's examine an example of a component which uses a ternary expression to randomly change the text that is displayed.

```
1
2
3
4
5
6
7

function Example() {
  return (
    <div className="heading">
      <h1>{Math.random() >= 0.5 ? "Over 0.5" : "Under 0.5"}</h1>
    </div>
  );
}
```

Inside the `<h1>` element, the curly braces signal to React that you want it to parse the code inside as regular JavaScript.

Then, inside the curly braces, you can add a ternary statement. Every ternary statement conceptually, expressed in pseudo-code, works like this:

```
1

comparison ? true : false
```

In the actual code example at the start of this lesson item, the comparison part, which goes to the left of the question mark, is using the `>=` (greater-than-or-equal-to operator), to return a Boolean

value. If the result of the comparison evaluates to `true`, then the string "Over 0.5" gets returned. In other words, whatever sits between the question mark and the semi-colon character will get returned. Otherwise, if the result of the comparison evaluates to `false`, then the string "Under 0.5" gets returned. In other words, the value that sits to the right of the colon character will get returned from the ternary expression.

This is how you can use a ternary expression to check for a condition right inside a component and return a value dynamically.

Using function calls in JSX

Another way to work with an expression in JSX is to invoke a function. Function invocation is an expression because every expression returns a value, and function invocation will always return a value, even when that return value is `undefined`.

Like the previous example, you can use function invocation inside JSX to return a random number:

```
1
2
3
4
5
6
7
8
9

function Example2() {
  return (
    <div className="heading">
      <h1>Here's a random number from 0 to 10:
        { Math.floor(Math.random() * 10) + 1 }
    </h1>
  )
}
```

```
</div>

) ;

} ;
```

In the `Example2` component, built-in `Math.floor()` and `Math.random()` methods are being used, as well as some number values and arithmetic operators, to display a random number between 0 and 10.

You can also extract this functionality into a separate function:

```
1
2
3
4
5
6
7
8
9
10

function Example3() {
  const getRandomNum = () => Math.floor(Math.random() * 10) + 1
  return (
    <div className="heading">
```

```
<h1>Here's a random number from 0 to 10: { getRandomNum()  
}</h1>  
  
</div>  
  
) ;  
  
};
```

The `getRandomNum()` function can also be written as a function declaration, or as a function expression. It does not have to be an arrow function.

But let's observe both alternatives: the function expression *and* the function declaration.

Function expression:

```
1  
  
2  
  
3  
  
const getRandomNum = function() {  
  
    return Math.floor(Math.random() * 10) + 1  
  
} ;
```

Function declaration:

```
1  
  
2  
  
3  
  
function getRandomNum() {  
  
    return Math.floor(Math.random() *10) + 1  
  
} ;
```

Of course, there are many other examples. The ones used here are there to help you understand how versatile and seamless the JSX syntax is. As you improve your React skills, you will find many creative ways of using JavaScript expressions in JSX.

Now that you have completed this reading, you have learned about a few more ways that you can use expressions in JSX.

Completed

Expressions as props

You've already learned a bit about using expressions as props. These can be, among other things, ternary operators, function calls, or some arithmetic operations.

However, you can pass almost any kind of expression as a prop.

For example:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

```

const bool = false;

function Example(props) {

    return (
        <h2>The value of the toggleBoolean prop is:
{props.toggleBoolean.toString()}</h2>
    );
}

export default function App() {
    return (
        <div className="App">
            <Example toggleBoolean={!bool} />
        </div>
    );
}

```

In the example above, you're using the `!bool`, that is, the NOT operator, which evaluates to `true`, since `!false` is true.

Also, for the `toggleBoolean` prop to be rendered on the page, you're converting its boolean value to a string using the JavaScript's built-in `toString` method.

Here's an extension of the above code which shows more ways to work with expressions as props in React.

What is happening here is several props are being passed to the `Example` component, and rendering each of these props' values to the screen.

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

```
const bool = false;

const str1 = "just";

function Example(props) {

  return (
    <div>
      <h2>
        The value of the toggleBoolean prop
        is:{props.toggleBoolean.toString()}

      </h2>
      <p>The value of the math prop is: <em>{props.math}</em></p>
      <p>The value of the str prop is: <em>{props.str}</em></p>
    </div>
  );
}

export default function App() {
```

```

    return (
      <div className="App">
        <Example
          toggleBoolean={!bool}
          math={(10 + 20) / 3}
          str={str1 + ' another ' + 'string'}
        />
      </div>
    );
  };

```

In this improvement to the `Example` component, three props are being passed to it: `toggleBoolean`, `math`, and `str`. The `toggleBoolean` is unchanged, and the `math` prop and the `str` prop have been added.

The `math` prop is there to show that you can add arithmetic operators and numbers inside JSX, and it will be evaluated just like it does in plain JavaScript.

The `str` prop is there to show that you can concatenate strings, as well as strings and variables – which is shown by adding string literals of “another” and “string” to the `str1` variable.

In summary, just like you can use expressions inside function components, you can also use them as prop values inside JSX elements, when rendering those function components.

Completed

Solution: Multiple components

Here's the completed App.js file:

```
2
3
4
5
6
7
8
9
10
11
12
13
14
15

import "./App.css";

import Card from "./Card";

function App() {
  return (
    <div className="App">
      <h1>Task: Add three Card elements</h1>
    </div>
  );
}

export default App;
```

```
<Card h2="First card's h2" h3="First card's h3" />

<Card h2="Second card's h2" h3="Second card's h3" />

<Card h2="Third card's h2" h3="Third card's h3" />

</div>

);

};

export default App;
```

Here's the completed Card.js file:

```
1
2
3
4
5
6
7
8
9
10

function Card(props) {
  return (
    <div>
      <Card h2="First card's h2" h3="First card's h3" />
      <Card h2="Second card's h2" h3="Second card's h3" />
      <Card h2="Third card's h2" h3="Third card's h3" />
    </div>
  );
}

export default App;
```

```
<div className="card">

  <h2>{props.h2}</h2>

  <h3>{props.h3}</h3>

</div>

) ;

};

export default Card;
```

Here is the output from the completed solution code:

← → C 🔒 ebyjtcchc.labs.coursera.org/serve/

Task: Add three Card elements

The image shows three separate rectangular boxes, each representing a card. The first card contains the text "First card's h2" and "First card's h3". The second card contains "Second card's h2" and "Second card's h3". The third card contains "Third card's h2" and "Third card's h3". Each card is enclosed in a thin gray border.

Step 1. Your first task was to add a Card.js file, and inside of that file, declare a `card` function.

1

2

3

```
function Card(props) {  
}  
};
```

Step 2. Inside the Card.js file's `card` function, you added a `return` statement with two parentheses after it, to allow the `return` statement to spread onto several lines.

```
1  
2  
3  
4  
5  
  
function Card(props) {  
  return (  
    );  
};
```

Step 3. Inside the `return` statement, you needed to add a wrapping `div` element.

```
1  
2  
3  
4  
5  
6  
  
function Card(props) {  
  return (  
    );  
};
```

```
    return (
      <div>
        </div>
      ) ;
  } ;
```

Step 4. Inside the wrapping `div` element, you needed to add an `h2` element, and under it, an `h3` element.

```
1
2
3
4
5
6
7
8
```

```
function Card(props) {
  return (
    <div>
      <h2></h2>
      <h3></h3>
    </div>
  ) ;
```

```
};
```

Step 5. Inside the `h2` element you've already added, you needed to add the JSX expression of: `{props.h2}`.

```
1  
2  
3  
4  
5  
6  
7  
8  
  
function Card(props) {  
  
  return (  
  
    <div className="card">  
  
      <h2>{props.h2}</h2>  
  
      <h3></h3>  
  
    </div>  
  
  );  
  
};
```

Step 6. Inside the `h3` element you've already added, you needed to add the JSX expression of: `{props.h3}`.

```
1  
2
```

```
3
4
5
6
7
8
9

function Card(props) {
  return (
    <div className="card">
      <h2>{props.h2}</h2>
      <h3>{props.h3}</h3>
    </div>
  );
}
```

Step 7. You also had to make sure to not forget the `export default Card;` line of code at the bottom of the Card component.

1

2

3

4

```
5
6
7
8
9
10

function Card(props) {
  return (
    <div className="card">
      <h2>{props.h2}</h2>
      <h3>{props.h3}</h3>
    </div>
  );
}

export default Card;
```

Step 8. Back inside the App component, you needed to import the `Card` component.

```
1
2
3
4
```

```
5
6
7
8
9
10
11
12

import "./App.css";

import Card from "./Card";

function App() {
  return (
    <div>
      <h1>Task: Add three Card elements</h1>
    </div>
  );
}

export default App;
```

Step 9. Inside the App component's `return` statement, under the `h1` element, you should have added three `<Card />` JSX elements.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
import "./App.css";
import Card from "./Card";
function App() {
  return (
    <h1>React</h1>
    <Card></Card>
    <Card></Card>
    <Card></Card>
  )
}

export default App;
```

```
<div>

  <h1>Task: Add three Card elements</h1>

  <Card />

  <Card />

  <Card />

</div>

);

};

export default App;
```

Step 10. You needed to add the `h2` prop to the first `<Card />` element, like this: `h2="First card's h2"`.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

```
import './App.css';

import Card from './Card';

function App() {
  return (
    <div>
      <h1>Task: Add three Card elements</h1>
      <Card h2="First card's h2" />
      <Card />
      <Card />
    </div>
  );
}

export default App;
```

Step 11. You needed to add the `h3` prop to the first `<Card />` element, like this: `h3="First card's h3"`.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
import "./App.css";
import Card from "./Card";
function App() {
  return (
    <div>
      <Card h3="First card's h3">
        <h3>First card</h3>
        <p>This is the first card. It has a title and some text.</p>
      </Card>
      <Card h3="Second card's h3">
        <h3>Second card</h3>
        <p>This is the second card. It also has a title and some text.</p>
      </Card>
    </div>
  );
}

export default App;
```

```
<div>

  <h1>Task: Add three Card elements</h1>

  <Card h2="First card's h2" h3="First card's h3" />

  <Card />

  <Card />

</div>

) ;

};

export default App;
```

Step 12. You should have added the `h2` and `h3` props to the second `<Card />` element, with the `h2` reading: "`Second card's h2`", and the `h3` reading "`Second card's h3`".

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

```
import './App.css';

import Card from './Card';

function App() {
  return (
    <div>
      <h1>Task: Add three Card elements</h1>
      <Card h2="First card's h2" h3="First card's h3" />
      <Card h2="Second card's h2" h3="Second card's h3" />
      <Card />
    </div>
  );
}

export default App;
```

Step 13. You should have added the `h2` and `h3` props to the third `<Card />` element, with the `h2` reading: "Third card's h2", and the `h3` reading "Third card's h3".

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

import "./App.css";

import Card from "./Card";

function App() {

  return (
    <div>
      <Card h2="First card's h2" h3="First card's h3">
        <p>This is the first card.</p>
      </Card>
      <Card h2="Second card's h2" h3="Second card's h3">
        <p>This is the second card.</p>
      </Card>
      <Card h2="Third card's h2" h3="Third card's h3">
        <p>This is the third card.</p>
      </Card>
    </div>
  );
}

export default App;
```

```
<div>

  <h1>Task: Add three Card elements</h1>

  <Card h2="First card's h2" h3="First card's h3" />

  <Card h2="Second card's h2" h3="Second card's h3" />

  <Card h2="Third card's h2" h3="Third card's h3" />

</div>

);

};

export default App;
```

Step 14. You needed to add the `className` attribute to the Card.js file's `Card` function's `return` statement's wrapping `div` element.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

```
import './App.css';

import Card from './Card';

function App() {
  return (
    <div className="App">
      <h1>Task: Add three Card elements</h1>
      <Card h2="First card's h2" h3="First card's h3" />
      <Card h2="Second card's h2" h3="Second card's h3" />
      <Card h2="Third card's h2" h3="Third card's h3" />
    </div>
  );
}

export default App;
```

Completed

Additional resources

Below you will find links to helpful additional resources.

- [Components and props](#)
- [Introducing JSX](#)
- [Styling and CSS in React](#)
- [Introducing expressions in JSX](#)

Completed

Week2-Eventful issues

You're now aware that React can work with most of the same events found in HTML, although React handles them differently.

This means that you may encounter unfamiliar errors when you run your event-driven React code. However, in this reading, you'll learn about some of the most common errors associated with events and how you can deal with them.

Event Errors

When you work in any programming environment, language, or framework, you are bound to write code that throws errors, for a variety of reasons.

Sometimes it's just about writing the wrong syntax. Other times it's about not thinking of all the possible scenarios and all the possible ways that things can go wrong in your code.

Regardless of what causes them, errors are a part of everyday life for a developer.

The JavaScript language comes with a built-in error handling syntax, the **try...catch** syntax.

Let's examine an example of an error in JavaScript:

1

```
(5) .toUpperCase()
```

Obviously, you cannot uppercase a number value, and thus, this throws the following error:

1

```
Uncaught TypeError: 5.toUpperCase is not a function
```

To handle this TypeError, you can update the code with a try...catch block that instructs the code to continue running after the error is encountered:

1

2

3

4

5

6

7

```
try {
  (5).toUpperCase();
}

catch(e) {
  console.log(`Oops, you can't uppercase a number.

  Trying to do it resulted in the following`, e);
}
```

The try-catch block will output some text in the console:

Oops, you can't uppercase a number. Trying to do it resulted in the following TypeError:

5.toUpperCase is not a function

It is assumed that if you are taking this course that you are already familiar with how the try...catch syntax works, so I won't go into any details after this quick refresher.

Back to React, here's an example of a simple error in a React component:

1

```
2
3
4
5
6
7
8
9

function NumBillboard(props) {
  return (
    <>
    <h1>{prop.num}</h1>
    </>
  )
}

export default NumBillboard;
```

In React, an error in the code, such as the one above, will result in the error overlay showing in the app in the browser.

In this specific example, the error would be:

ReferenceError

prop is not defined

Note: *You can click the X button to close the error overlay.*

Since event-handling errors occur after the UI has already been rendered, all you have to do is use the error-handling mechanism that already exists in JavaScript – that is, you just use the `try...catch` blocks.

Completed

Event handling and embedded expressions

In this reading, you'll learn the different ways to embed expressions in event handlers in React:

- With an inline anonymous ES5 function
- With an inline, anonymous ES6 function (an arrow function)
- Using a separate function declaration
- Using a separate function expression

You may find this reading useful as a reference sheet.

For clarity and simplicity: a function will simply console log some words. This will allow you to compare the difference in syntax between these four approaches, while the result of the event handling will always be the same: just some words output to the console.

Handling events using inline anonymous ES5 functions

This approach allows you to directly pass in an ES5 function declaration as the `onClick` event-handling attribute's value:

1

2

3

```
<button onClick={function() {console.log('first example')}}>  
  An inline anonymous ES5 function event handler  
</button>
```

Although it's possible to write your click handlers using this syntax, it's not a common approach and you will not find such code very often in React apps.

Handling events using inline anonymous ES6 functions (arrow functions)

With this approach, you can directly pass in an ES6 function declaration as the `onClick` event-handling attribute's value:

```
1 <button onClick={() => console.log('second example')}>  
2   An inline anonymous ES6 function event handler  
3 </button>
```

This approach is much more common than the previous one. If you want to keep all your logic inside the JSX expression assigned to the `onClick` attribute, use this syntax.

Handling events using separate function declarations

With this approach, you declare a separate ES5 function declaration, and then you reference its name in the event-handling `onClick` attribute, as follows:

```
1 <button onClick={handleClick}>  
2   A separate function declaration event handler  
3 </button>  
4  
5 function handleClick() {  
6   console.log('third example')  
7 }  
8  
9
```

10

11

12

13

```
function App() {  
  
  function thirdExample() {  
  
    console.log('third example');  
  
  };  
  
  return (  
  
    <div className="thirdExample">  
  
      <button onClick={thirdExample}>  
  
        using a separate function declaration  
  
      </button>  
  
    </div>  
  
  );  
  
}  
  
export default App;
```

This syntax makes sense to be used when your onClick logic is too complex to easily fit into an anonymous function. While this example is not really showing this scenario, imagine a function that has, for example, 20 lines of code, and that needs to be ran when the click event is triggered. This is a perfect use-case for a separate function declaration.

Handling events using separate function expressions

Tip: A way to determine if a function is defined as an expression or a declaration is: if it does not start the line with the keyword `function`, then it's an expression.

In the following example, you're assigning an anonymous ES6 arrow function to a `const` variable – hence, this is a function expression.

You're then using this `const` variable's name to handle the `onClick` event, so this is an example of handling events using a separate function expression.

```
1
2
3
4
5
6
7
8
9
10
11
12

function App() {
  const fourthExample = () => console.log('fourth example');

  return (
    <div className="fourthExample">
      <button onClick={fourthExample}>
```

```
        using a separate function expression

    </button>

</div>

);

};

export default App;
```

The syntax in this example is very common in React. It uses arrow functions, but also allows us to handle situations where our separate function expression spans multiple lines of code.

In this reading lesson item, you've learned the several types of functions you can use to handle events in React. Some of those are more common than others, but now that you know all the different ways of doing this, you can understand other people's code more easily, as well as choose the syntax that best suits your given use case, such as a specific company coding style guide.

Completed

Solution: Dynamic events

Here is the completed App.js file:

1

2

3

4

5

6

7

```
8
9
10
11
12
13
14
15
16
17
18

function App() {
  const [randomNum, setRandomNum] = useState(Math.floor(Math.random() * 3) + 1);
  const [userInput, setUserInput] = useState('');
  const [guessStatus, setGuessStatus] = useState('');

  function handleClick() {
    let randomNum = Math.floor(Math.random() * 3) + 1;
    console.log(randomNum);

    let userInput = prompt('type a number');

    alert(`Computer number: ${randomNum}, Your guess: ${userInput}`);
  }

  return (
    <div>
      <p>Computer number: {randomNum}</p>
      <p>Your guess:</p>
      <input type="text" value={userInput} onChange={(e) => setUserInput(e.target.value)} />
      <button onClick={handleClick}>Check</button>
      <p>Status: {guessStatus}</p>
    </div>
  );
}
```

```

<div>

  <h1>Task: Add a button and handle a click event</h1>

  <button onClick={handleClick}>Guess the number between 1 and
3</button>

</div>

);

}

export default App;

```

Here is the output from the solution code for the App.js file:

Task: Add a button and handle a

awdywyxz.labs.coursera.org says
Computer number: 2, Your guess: 3

Step 1. First, you added a `button` element, with an opening and a closing `button` tag, to the `App` component's `h1` element .

1

2

3

4

5

6

7

8

9

10

11

```
function App() {  
  
    return (  
        <div>  
            <h1>Task: Add a button and handle a click event</h1>  
            <button></button>  
        </div>  
    );  
  
}  
  
export default App;
```

Step 2. In between the opening and closing `button` tags, you added the following text: Guess the number between 1 and 3.

1

2

3

4

5

6

7

```
8  
9  
10  
11  
  
function App() {  
  
  return (  
    <div>  
      <h1>Task: Add a button and handle a click event</h1>  
      <button>Guess the number between 1 and 3</button>  
    </div>  
  );  
  
}  
  
export default App;
```

Step 3. Next, inside the opening `button` tag, you added the `onClick` event-handling attribute, and passed it the following JSX expression: `{handleClick}`.

1
2
3
4
5

```
6
7
8
9
10
11

function App() {
  return (
    <div>
      <h1>Task: Add a button and handle a click event</h1>
      <button onClick={handleClick}>Guess the number between 1 and 3</button>
    </div>
  );
}

export default App;
```

Step 4. Then, above the `return` statement of the App component - but still inside the App function - you added the following ES5 function declaration:

1

2

```
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

function App() {
    let randomNum = Math.floor(Math.random() * 3) + 1;
    console.log(randomNum);
}

function handleClick() {
    let randomNum = Math.floor(Math.random() * 3) + 1;
    console.log(randomNum);
}
```

```

let userInput = prompt('type a number');

alert(`Computer number: ${randomNum}, Your guess: ${userInput}`);

}

return (
<div>

<h1>Task: Add a button and handle a click event</h1>

<button onClick={handleClick}>Guess the number between 1 and
3</button>

</div>
);
}

export default App;

```

Step 5. Finally, you saved your changes and ran the app to preview it in the browser. You should then be able to click a button, which will show a prompt pop up which you can type into. After that, an alert pop up will show computer's "choice" and your guess. After you click "ok" to close the alert, you'll be able to click the button again and try matching the number "chosen" by the computer one more time.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

- [Handling Events](#)

- [Supported Events](#)
- [SyntheticEvent](#)
- [How do I pass an event handler \(like onClick\) to a component?](#)
- [JavaScript Expressions as Props](#)
- [JavaScript Expressions as Children](#)

Completed

ata flow in React

You've just learned how the parent-child relationship can be set up so that data flows from parent to child.

In this reading, you'll learn how to detail the flow of data from parent to child. You will then learn why code samples need to be clear and concise. Finally, you will explore data flow in greater detail by looking at more examples. This should act as a refresher to knowledge gained in previous courses.

Parent-child data flow

In React, data flow is a one-way street. Sometimes it's said that the data flow is unidirectional. Put differently, the data in React flows from a parent component to a child component. The data flow starts at the root and can flow to multiple levels of nesting, from the root component (parent component) to the child component, then the grandchild component, and further down the hierarchy. A React app consists of many components, organized as a component tree. The data flows from the root component to all the components in the tree structure that require this data, using props.

Props are immutable (cannot be changed).

The two main benefits of this unidirectional data flow are that it allows developers to:

1. comprehend the logic of React apps more quickly and
2. simplify the data flow.

Here's a practical example of this:

Imagine that the parent component passes a prop (name) to the child component. The child component then uses this prop to render the name in the UI.

Parent component:

1

2

3

4

```
function Dog() {  
  
  return (  
  
    <Puppy name="Max" bowlShape="square" bowlStatus="full" />  
  
  );  
  
};
```

Child component:

```
1  
  
2  
  
3  
  
4  
  
5  
  
6  
  
7  
  
function Puppy(props) {  
  
  return (  
  
    <div>  
  
      {props.name} has <Bowl bowlShape="square" bowlStatus="full" />  
  
    </div>  
  
  );  
  
};
```

Grandchild component:

```
1  
2  
3  
4  
5  
6  
7  
  
function Bowl(props) {  
  
  return (  
  
    <span>  
  
      {props.bowlShape}-shaped bowl, and it's currently  
      {props.bowlStatus}  
  
    </span>  
  
  );  
  
}  
;
```

Having data move through props in only one direction makes it simpler to understand the logic of how the components interact. If data were moving everywhere, all the time, then it would be much harder to comprehend its logical flow. Any optimization you tried to implement would likely not be as efficient as it could be, especially in modern React.

Completed

Using hooks

Now that you understand what hooks are in React and have some basic knowledge on the `useState` hook, let's dive in deeper. In this reading, you will learn how to use hooks in React components and understand the use-cases for the `useState` hook.

Let's say you have a component with an input text field. The user can type into this text field. The component needs to keep track of what the user types within this text field. You can add state and use the `useState` hook, to hold the string.

As the user keeps typing, the local state that holds the string needs to get updated with the latest text that has been typed.

Let's discuss the below example.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

```
import { useState } from 'react';

export default function InputComponent() {
  const [inputText, setText] = useState('hello');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <>
      <input value={inputText} onChange={handleChange} />
      <p>You typed: {inputText}</p>
      <button onClick={() => setText('hello')}>
        Reset
      </button>
    </>
  );
}
```

```
) ;  
}
```

To do this, let's define a React component and call it `InputComponent`. This component renders three things:

- An input text field
- Any text that has been entered into the field
- A Reset button to set the field back to its default state

As the user starts typing within the text field, the current text that was typed is also displayed.



You typed: Welcome



The state variable `inputText` and the `setText` method are used to set the current text that is typed. The `useState` hook is initialized at the beginning of the component.

1

```
const [inputText, setText] = useState('hello');
```

By default, the `inputText` will be set to "hello".

As the user types, the `handleChange` function, reads the latest input value from the browser's input DOM element, and calls the `setText` function, to update the local state of `inputText`.

1

2

3

```
function handleChange(e) {  
  
  setText(e.target.value);  
  
};
```

Finally, clicking the reset button will update the `inputText` back to “hello”.

Isn’t this neat?

Keep in mind that the `inputText` here is local state and is local to the `InputComponent`. This means that outside of this component, `inputText` is unavailable and unknown. In React, state is always referred to the local state of a component.

Hooks also come with a set of rules, that you need to follow while using them. This applies to all React hooks, including the `useState` hook that you just learned.

- You can only call hooks at the top level of your component or your own hooks.
- You cannot call hooks inside loops or conditions.
- You can only call hooks from React functions, and not regular JavaScript functions.

To demonstrate, let’s extend the previous example, to include three input text fields within a single component. This could be a registration form with fields for first name, last name and email.

First name:

Last name:

Email:

Luke Jones (lukeJones@sculpture.com)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

```
31
32
33
34
35
36
37
38
39
40

import { useState } from 'react';

export default function RegisterForm() {
  const [form, setForm] = useState({
    firstName: 'Luke',
    lastName: 'Jones',
    email: 'lukeJones@sculpture.com',
  });

  return (
    <>

```

```
<label>

    First name:

    <input
        value={form.firstName}

        onChange={e => {
            setForm({
                ...form,
                firstName: e.target.value
            });
        }}
    />

</label>

<label>

    Last name:

    <input
        value={form.lastName}

        onChange={e => {
            setForm({
                ...form,
                lastName: e.target.value
            });
        }}
    />
```

```

        } }

    />

</label>

<label>

  Email:

<input

  value={form.email}

  onChange={e => {

```

Notice that you are using a `form` object to store the state of all three text input field values:

```

  1

  2

  3

  4

  5

const [form, setForm] = useState({
  firstName: 'Luke',
  lastName: 'Jones',
  email: 'lukeJones@sculpture.com',
}) ;

```

You do not need to have three separate state variables in this case, and instead you can consolidate them all together into one `form` object for better readability.

In addition to the `useState` hook, there are other hooks that come in handy such as `useContext`, `useMemo`, `useRef`, etc. When you need to share logic and reuse the same logic across several

components, you can extract the logic into a custom hook. Custom hooks offer flexibility and can be used for a wide range of use-cases such as form handling, animation, timers, and many more. Next, I'll give you an explanation of how the `useRef` hook works.

The `useRef` hook

We use the `useRef` hook to access a child element directly.

When you invoke the `useRef` hook, it will return a `ref` object. The `ref` object has a property named `current`.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14

function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // ...
  };
  return (
    <div>
      <input type="text" ref={inputEl} />
      <button onClick={onButtonClick}>Click me</button>
    </div>
  );
}
```

```

// `current` points to the mounted text input element

inputEl.current.focus();

};

return (
<>
<input ref={inputEl} type="text" />

<button onClick={onButtonClick}>Focus the input</button>

</>

);
}

```

Using the `ref` attribute on the input element, I can then access the current value and invoke the `focus()` method on it, thereby focusing the input field.

There are situations where accessing the DOM directly is needed, and this is where the `useRef` hook comes into play.

Conclusion

In this reading, you have explored hooks in detail and understand how to use the `useState` hook to maintain state within a component. You also understand the benefits of using hooks within a React component.

Completed

Prop drilling

As you've learned previously, prop drilling is a situation where you are passing data from a parent to a child component, then to a grandchild component, and so on, until it reaches a more distant component further down the component tree, where this data is required.

Here is a very simple app that focuses on the process of props passing through several components.

Please note that the goal here is not to build an app that would exist in the real world. The goal of this app is to examine the practice of prop drilling, so that you can focus on it and understand it in isolation.

Here is the code for the app:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
function Main(props) {  
  
  return <Header msg={props.msg} />;  
}  
  
function Header(props) {  
  
  return (  
  
    <div style={{ border: "10px solid whitesmoke" }}>  
  
      <h1>Header here</h1>  
  
      <Wrapper msg={props.msg} />  
  
    </div>  
  );  
};  
  
function Wrapper(props) {  
  
  return (  
  
    <div style={{ border: "10px solid lightgray" }}>  
  
      <h2>Wrapper here</h2>  
  
      <Button msg={props.msg} />  
  
    </div>  
  );  
};
```

```
);

};

function Button(props) {
    return (
        <div style={{ border: "20px solid orange" }}>
            <h3>This is the Button component</h3>
            <button onClick={() => alert(props.msg)}>Click me!</button>
        </div>
    );
}

function App() {
    return (
        <Main
            msg="I passed through the Header and the Wrapper and I reached the
Button component"
        />
    );
}
```

```
export default App;
```

This app is simple enough that you should be able to understand it on your own. Let's address the main points to highlight what is happening in the code above.

The top-most component of this app is the `App` component. The `App` component returns the `Main` component. The `Main` component accepts a single attribute, named `msg`, as in “message”.

At the very top of the app, the `Main` function declares how the `Main` component should behave. The `Main` component is responsible for rendering the `Header` component. **Note that when the `Header` component is rendered from inside `Main`, it also receives the `msg` prop.**

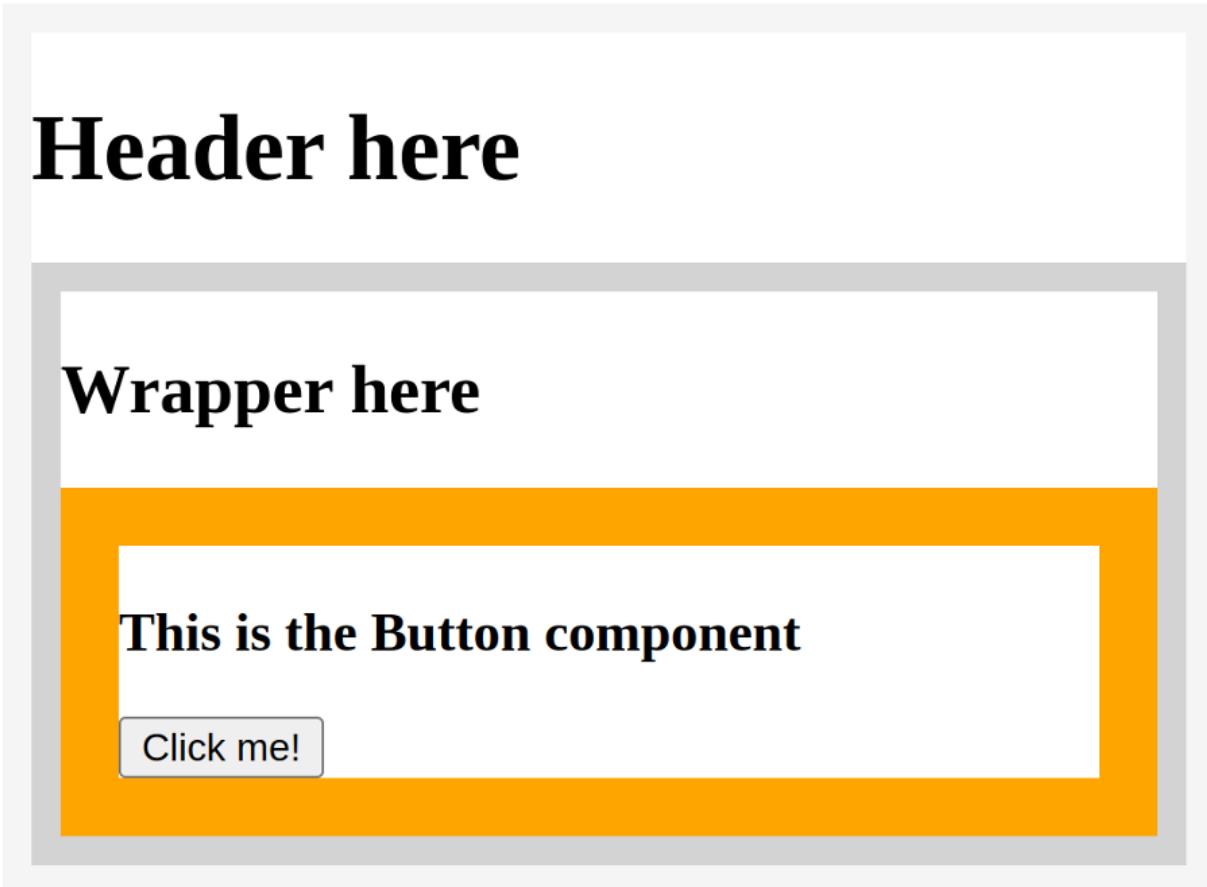
The `Header` component's function declaration renders an `h1` that reads “Header here”, then another component named `wrapper`. Note that the naming here is irrelevant – the components `Header` and `Wrapper` are named to make it a bit more like it might appear in a real app – but ultimately, the focus is on having multiple components, rather than describing specific component names properly.

So, the `Header` component's function declaration has a return statement, which **renders the `Wrapper` component with the `msg` prop passed to it**.

In the `Wrapper` component's function declaration, there's an `h2` that reads “Wrapper here”, in addition to **the rendering of the `Button` component, which also receives the `msg` attribute**.

Finally, the `Button` component's function declaration is coded to receive the `props` object, then inside of the wrapping `div`, show an `h3`. The `h3` reads “This is the Button component”, and then, under that, there's a button element with an `onClick` event-handling attribute. This is passed to an arrow function which should alert the string that comes from the `props.msg` prop.

All this code results in the following UI rendered on the screen:



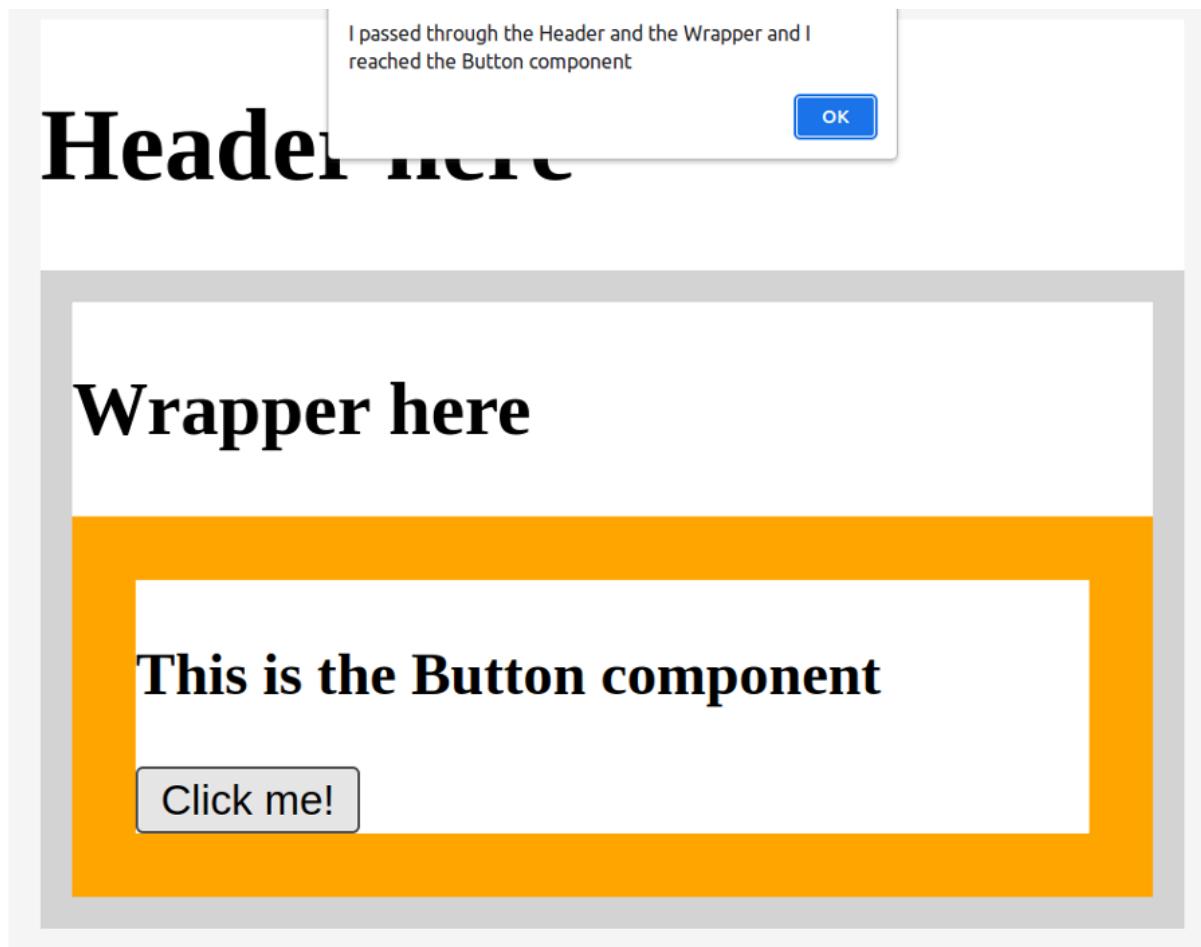
Header here

Wrapper here

This is the Button component

Click me!

This screenshot illustrates the boundaries of each component. The `Main` component can't be found in the UI because it's just rendering the `Header` component. The `Header` component then renders the `Wrapper` component, and the `Wrapper` component then renders the `Button` component. Note that the string that was passed on and on through each of the children component's props' objects is not found anywhere. However, it will appear when you click the "Click me!" button, as an alert:



The alert's message reads "I passed through the Header and the Wrapper and I reached the Button component".

That's really all there is to it. Props drilling simply means passing a prop through props objects through several layers of components. The more layers there are, the more repetitive and unnecessary this feels. There are various ways to deal with this, as you'll learn in the lesson items that follow.

Completed

Solution: Managing state in React

Here is the completed App.js file:

1

2

```
import React from "react";
```

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

```
import Fruits from "./Fruits";
import FruitsCounter from "./FruitsCounter";


function App() {
  const [fruits] = React.useState([
    {fruitName: 'apple', id: 1},
    {fruitName: 'apple', id: 2},
    {fruitName: 'plum', id: 3},
  ]);

  return (
    <div className="App">
      <h1>Where should the state go?</h1>
      <Fruits fruits={fruits} />
      <FruitsCounter fruits={fruits} />
    </div>
  );
}

export default App;
```

Here is the completed Fruits.js file:

```
1
2
3
4
5
6
7
8
9

function Fruits(props) {
  return (
    <div>
      {props.fruits.map(f => <p key={f.id}>{f.fruitName}</p>) }
    </div>
  )
}

export default Fruits
```

Here is the completed FruitsCounter.js file:

```
1
2
```

```
3
4
5
6
7

function FruitsCounter(props) {
  return (
    <h2>Total fruits: {props.fruits.length}</h2>
  )
}

export default FruitsCounter;
```

The completed app should look as follows:

Where should the state go?

```
apple
apple
plum
```

Total fruits: 3

Step 1. Move the state from the `Fruits` component to the `App` component.

To complete this step, you need to go to the `Fruits` component and cut the `useState` call, namely this piece of code:

1

2

3

4

5

6

7

```
const [fruits] = React.useState([
  {fruitName: 'apple', id: 1},
  {fruitName: 'apple', id: 2},
  {fruitName: 'plum', id: 3},
]);
```

You also need to cut the `import React from "react";` at the very top of the `Fruits` component, since you no longer need to access the `useState` method on the `React` object from the `Fruits` file. Additionally, you need to add the import statement to the `App` component, which means that you should inject a new import at the very top of `App.js`:

1

```
import React from "react";
```

Once you've done that, you need to update the `App` component's return statement so that it sends the fruits data to the `Fruits` and `FruitsCounter` component - since both of these components need to get this state's data via props.

1

2

3

4

5

6

7

8

// The updated return statement in `App.js`:

```
return (
```

```

<div className="App">

  <h1>Where should the state go?</h1>

  <Fruits fruits={fruits} />

  <FruitsCounter fruits={fruits} />

</div>

);

```

Step 2.

The `Fruits` component should be updated so that it accepts state from the `App` component. Now all that you need to do is to update the code in the `Fruits` components to accept the `props` object and render the `fruits` property where appropriate. That means that the `Fruits` component will end up having the following code:

```

1
2
3
4
5
6
7
8
9

function Fruits(props) {
  return (
    <div>
      {props.fruits.map(f => <p key={f.id}>{f.fruitName}</p>) }
    </div>
  );
}

```

```
</div>

)

}

export default Fruits
```

Step 3.

Once you've lifted the state up from the `Fruits` component to the `App` component, you also need to update the `FruitsCounter` component.

Just like the `Fruits` component, the `FruitsCounter` component should also receive state from the `App` component, so that it can display the number of the available fruits using the `length` property of the array of fruits from the `fruits` state variable.

The `FruitsCounter` component will end up having the following code:

```
1

2

3

4

5

6

7

function FruitsCounter(props) {

  return (
    <h2>Total fruits: {props.fruits.length}</h2>
  )
}
```

```
export default FruitsCounter;
```

That completes this ungraded lab's solution.

Completed

Additional resources

Below you will find links to helpful additional resources.

- [React docs website URL which discusses the issue in depth](#)
- [Data flows down](#)
- [The Power Of Not Mutating Data](#)
- [Add Inverse Data Flow](#)
- [Component state](#)
- [State: A Component's Memory](#)
- [Sharing State Between Components](#)
- [State as a Snapshot](#)
- [Basic useState examples](#)
- [Synchronizing with effects - putting it all together](#)
- [Fetch API](#)
- [The event loop in JavaScript](#)

Completed

Week3-

Navigation

In this reading, you'll learn about the differences between traditional web pages and React-powered web pages (SPAs – single page applications).

Once you understand the difference between these two ways of building web pages, you will be able to understand the necessary difference between how navigation works in traditional web apps versus how it works in modern SPA websites.

Before Single-Page Apps

Before the advent of modern JavaScript frameworks, most websites were implemented as multi-page applications. That is, when a user clicks on a link, the browser navigates to a new webpage, sends a request to the web server; this then responds with the full webpage and the new page is displayed in the browser.

This can make your application resource intensive to the Web Server. CPU time is spent rendering dynamic pages and network bandwidth is used sending entire webpages back for every request. If your website is complex, it may appear slow to your users, even slower if they have a slow or limited internet connection.

To solve this problem, many web developers develop their web applications as Single Page Applications.

Single-Page Apps

You're using many Single Page Applications every day. Think of your favorite social network, or online email provider, or the map application you use to find local businesses. Their excellent user experiences are driven by Single Page Applications.

A Single Page Application allows the user to interact with the website without downloading entire new webpages. Instead, it rewrites the current webpage as the user interacts with it. The outcome is that the application will feel faster and more responsive to the user.

How Does a Single-Page App Work?

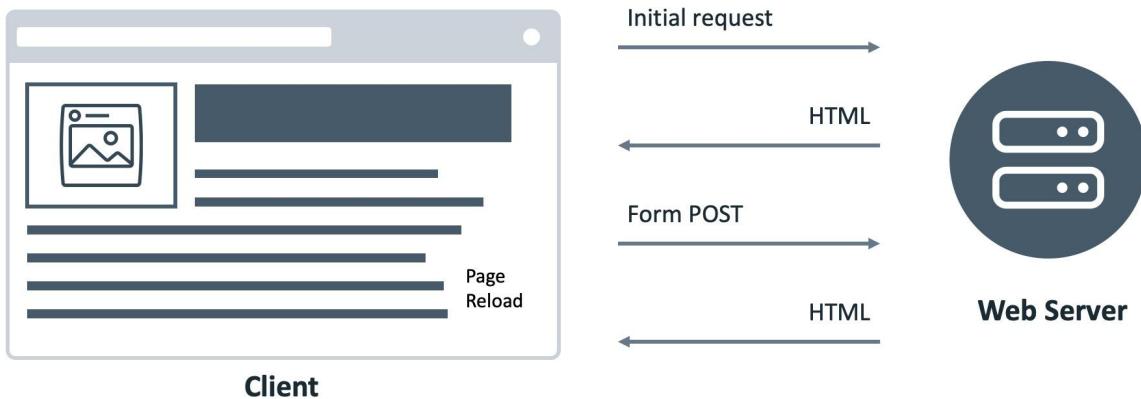
When the user navigates to the web application in the browser, the Web Server will return the necessary resources to run the application. There are two approaches to serving code and resources in Single Page Applications.

1. When the browser requests the application, return and load all necessary HTML, CSS and JavaScript immediately. This is known as *bundling*.
2. When the browser requests the application, return only the minimum HTML, CSS and JavaScript needed to load the application. Additional resources are downloaded as required by the application, for example, when a user navigates to a specific section of the application. This is known as *lazy loading* or *code splitting*.

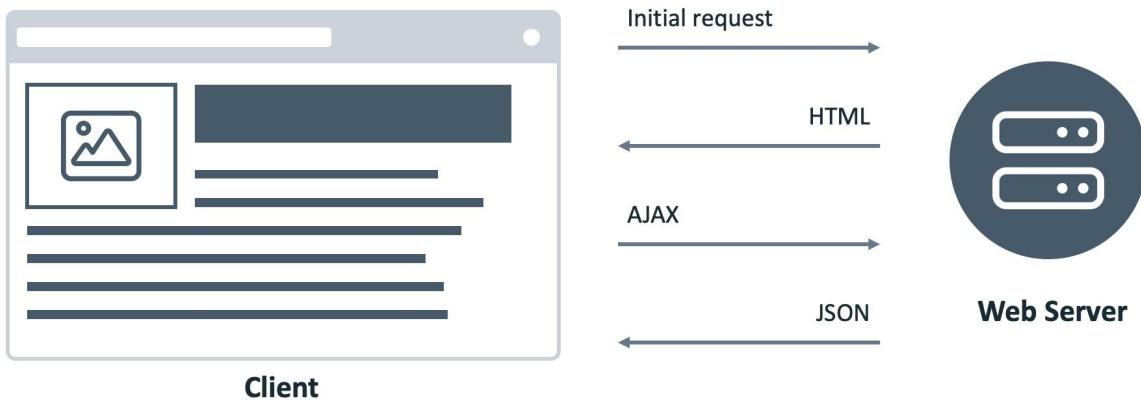
Both approaches are valid and are used depending on the size, complexity and bandwidth requirements of the application. If your application is complex and has a lot of resources, your bundles will grow quite large and take a long time to download – possibly ending up slower than a traditional web application!

Once the application is loaded, all logic and changes are applied to the current webpage. Let's look at an example.

Traditional Page Lifecycle



SPA Page Lifecycle



An Example of a Single-Page App

Imagine there is a webpage that has a Label and a Button. It will display a random movie name when the button is clicked.

In a traditional website, when the button is clicked, the browser will send a POST request to the web server. The web server will return a new web page containing the button and movie name, and the web browser renders the new page.

In a Single Page Application, when the button is clicked, the browser will send a POST request to a web server. The web server will return a JSON object. The application reads the object and updates the Label with the movie name.

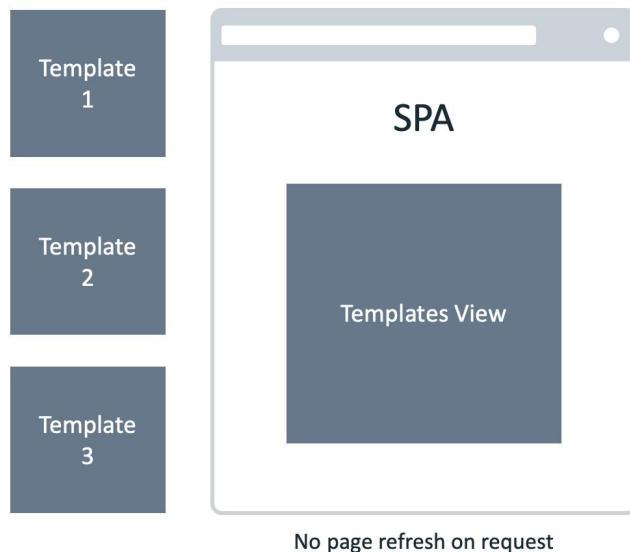
See, more efficient!

But what if we need to have multiple pages with different layouts in our application?

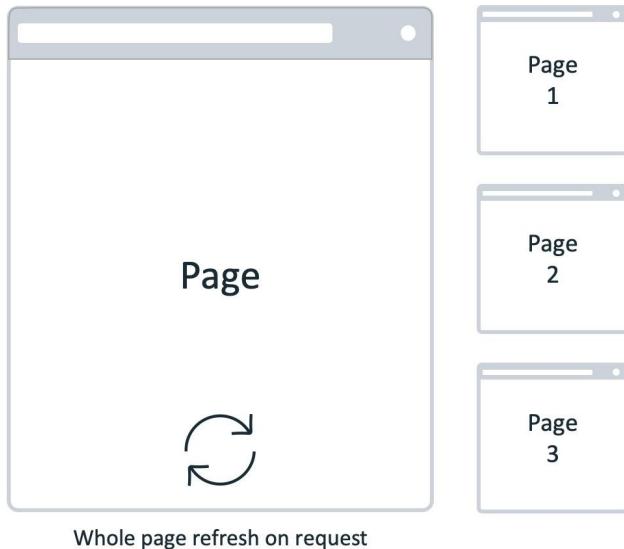
Let's look at another example.

Practical Differences Between Single-Page Apps and Multi-Page Apps

Single Page Application



Traditional Page Application



Whole page refresh on request

You have a web application that has a navigation bar on top and two pages. One page shows the latest news, and the other shows the current user's profile page. The navigation bar contains a link for each page.

In a traditional website, when the user clicks the Profile link, the web browser sends the request to the web server. The web server generates the HTML page and sends it back to the web browser. The web browser then renders the new web page.

In a Single Page Application, different pages are broken into templates (or views). Each view will have HTML code containing variables that can be updated by the application.

The web browser sends the request to the web server, and the web server sends back a JSON object. The web browser then updates the web page by inserting the template with the variables replaced by the values in the JSON object.

Anchor Tag Elements in Single-Page Elements

A single-page application can't have regular anchor tag elements as a traditional web app can.

The reason for this is that the default behavior of an anchor tag is to load another HTML file from a server and refresh the page. This page refresh is not possible in a SPA that's powered by a library such as React because a total page refresh is not the way that a SPA works, as explained earlier in this lesson item.

Instead, a SPA comes with its own special implementation of anchor tags and links, which only give an illusion of loading different pages to the end user when in fact, they simply load different components into a single element of the real DOM into which the virtual DOM tree gets mounted and updated.

That's why navigation in a single-page app is fundamentally different from its counterpart in a multi-page app. Understanding the concepts outlined in this lesson item will make you a more well-rounded React developer.

Completed

Solution: Creating a route

Here is the Contact.js file:

```
1
2
3
4

function Contact() {
  return <h1>Contact Little Lemon on this page.</h1>
}

export default Contact
```

Here is the completed App.js file:

```
1
2
3
4
5
6
7
8
9
```

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

import "./App.css";

import Homepage from "./Homepage";

import AboutLittleLemon from "./AboutLittleLemon";

import Contact from "./Contact";

import { Routes, Route, Link } from "react-router-dom";
```

```
function App() {  
  return (  
    <div>  
      <nav>  
        <Link to="/" className="nav-item">Homepage</Link>  
        <Link to="/about" className="nav-item">About Little Lemon</Link>  
        <Link to="/contact" className="nav-item">Contact</Link>  
      </nav>  
      <Routes>  
        <Route path="/" element={<Homepage />}></Route>  
        <Route path="/about" element={<AboutLittleLemon />}></Route>  
        <Route path="/contact" element={<Contact />}></Route>  
      </Routes>  
    </div>  
  );  
}  
  
export default App;
```

Here is the output from the completed solution code:

Welcome to the Little Lemon site

Step 1

First, you added a new file, Contact.js, to the root of the src folder.

Step 2

Inside the Contact.js file, you added an ES5 function, named **Contact**. And then, added the `export default Contact` after the Contact function's closing curly brace.

1

2

3

4

```
function Contact() {  
}  
  
export default Contact;
```

Step 3

Next, inside the body of the Contact function, you added a `return` statement with the following code: `<h1>Contact Little Lemon on this page.</h1>`.

1

2

3

4

```
function Contact() {  
  
  return  <h1>Contact Little Lemon on this page.</h1>  
  
};  
  
export default Contact;
```

Step 4

Inside the App.js file, you imported the newly-built Contact component.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

```
import "./App.css";

import Homepage from "./Homepage";

import AboutLittleLemon from "./AboutLittleLemon";

import Contact from "./Contact";

import { Routes, Route, Link } from "react-router-dom";



function App() {

  return (

    <div>

      <nav>

        <Link to="/" className="nav-item">Homepage</Link>

        <Link to="/about" className="nav-item">About Little Lemon</Link>

      </nav>

      <Routes>
```

```
<Route path="/" element={<Homepage />}></Route>

<Route path="/about" element={<AboutLittleLemon />}></Route>

</Routes>

</div>

) ;

} ;

export default App;
```

Step 5

Inside the App.js file's App function's return statement, locate the `nav` element, and inside of it, add another `<Link>` element, with the `to` attribute pointing to contact, the `className` set to "nav-item", and the the text inside the `Link` element's opening and closing tags set to `Contact`.

1

2

3

4

5

6

7

8

9

10

```
11
12
13
14
15
16
17
18
19
20
21
22
23
24

import "./App.css";

import Homepage from "./Homepage";
import AboutLittleLemon from "./AboutLittleLemon";
import Contact from "./Contact";
import { Routes, Route, Link } from "react-router-dom";

function App() {
```

```

    return (
      <div>
        <nav>
          <Link to="/" className="nav-item">Homepage</Link>
          <Link to="/about" className="nav-item">About Little Lemon</Link>
          <Link to="/contact" className="nav-item">Contact</Link>
        </nav>
        <Routes>
          <Route path="/" element={<Homepage />}></Route>
          <Route path="/about" element={<AboutLittleLemon />}></Route>
        </Routes>
      </div>
    );
  };
}

export default App;

```

Step 6

Inside the `Routes` element, add a third route, with the path attribute pointing to `"/contact"`, and the element attribute set to `{<Contact />}`.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

```
import "./App.css";

import Homepage from "./Homepage";

import AboutLittleLemon from "./AboutLittleLemon";

import Contact from "./Contact";

import { Routes, Route, Link } from "react-router-dom";

function App() {

  return (

    <div>

      <nav>

        <Link to="/" className="nav-item">Homepage</Link>

        <Link to="/about" className="nav-item">About Little Lemon</Link>

        <Link to="/contact" className="nav-item">Contact</Link>

      </nav>

      <Routes>

        <Route path="/" element={<Homepage />}></Route>

        <Route path="/about" element={<AboutLittleLemon />}></Route>

        <Route path="/contact" element={<Contact />}></Route>

      </Routes>

    </div>

  );
}
```

```
};
```

Step 7

You saved all your changes and viewed your updates in the served app. You should have had three links in the top navbar, and the third link should have been Contact. Once you clicked the link, the sentence "Contact Little Lemon on this page" should have replaced whatever other content was under the navbar previously.

Completed

Applying conditional rendering

State is all the data your app is currently working with. With this in mind, you can decide to conditionally render specific components in your app, based on whether specific state data has specific values. To make this possible, React works with the readily available JavaScript syntax and concepts.

Consider a minimalistic productivity app.

The app takes the client computer's current datetime, and based on the data, displays one of two messages on the screen:

1. For workdays, the message is: "Get it done"
1. For weekends, the message is: "Get some rest"

There are a few ways you can achieve this in React.

One approach would include setting a component for each of the possible messages, which means you'd have two components. Let's name them `workdays` and `weekends`.

Then, you'd have a `CurrentMessage` component, which would render the appropriate component based on the value returned from the `getDay()` function call.

Here's a simplified `CurrentMessage` component:

1

2

3

4

5

6

```

function CurrentMessage() {
  const day = new Date().getDay();
  if (day >= 1 && day <= 5) {
    return <Workdays />
  }
  return <Weekends />
}

```

Instead of calculating it directly, you could use some historical data instead, and perhaps get that data from a user via an input, from a parent component.

In that case, the `CurrentMessage` component might look like this:

```

1
2
3
4
5
6

function CurrentMessage(props) {
  if (props.day >= 1 && props.day <= 5) {
    return <Workdays />
  }
  return <Weekends />
}

```

Conditional rendering with the help of element variables

To further improve your `CurrentMessage` component, you might want to use element variables. This is useful in some cases, where you want to streamline your render code - that is, when you want to separate the conditional logic from the code to render your UI.

Here's an example of doing this with the `CurrentMessage` component:

```
1<div>
2  <CurrentMessage>
3    <div>
4      <div>{{ message }}</div>
5    </div>
6    <div>
7      <div>{{ message }}</div>
8    </div>
9  </CurrentMessage>
10 </div>
11 <CurrentMessage>
12   <div>
13     <div>{{ message }}</div>
14   </div>
15 </CurrentMessage>
16 <CurrentMessage>
17   <div>
18     <div>{{ message }}</div>
19   </div>
20 </CurrentMessage>
```

```
function CurrentMessage({day}) {  
  
  const weekday = (day >= 1 && day <= 5);  
  
  const weekend = (day >= 6 && day <= 7);  
  
  let message;  
  
  if (weekday) {  
  
    message = <Workdays />  
  
  } else if (weekend) {  
  
    message = <Weekends />  
  
  } else {  
  
    message = <ErrorComponent />  
  
  }  
  
  return (  
  
    <div>  
  
      {message}  
  
    </div>  
  
  )  
}
```

The output of the `CurrentMessage` component will depend on what the received value of the day variable is. On the condition of the day variable having the value of any number between 1 and 5 (inclusive), the output will be the contents of the `Workdays` component. Otherwise, on the condition of the day variable having the value of either 6 or 7, the output will be the contents of the `Weekends` component.

Conditional rendering using the logical AND operator

Another interesting approach in conditional rendering is the use of the logical `AND` operator `&&`. In the following component, here's how the `&&` operator is used to achieve conditional rendering:

```
1
2
3
4
5
6
7
8
9
10
11
12

function LogicalAndExample() {
  const val = prompt('Anything but a 0')

  return (
    <div>
      {val && val > 0 ? <p>It's a positive number!</p> : <p>It's not a positive number!</p>}
    </div>
  )
}
```

```

<div>

  <h1>Please don't type in a zero</h1>

  {val &&

    <h2>Yay, no 0 was typed in!</h2>

  }

</div>

)
}

}

```

There are a few things to unpack here, so here is the explanation of the `LogicalAndExample` component, top to bottom:

1. First, you ask the user to type into the prompt, specifying that you require anything other than a zero character; and you save the input into the `val` value.
2. In the return statement, an `h1` heading is wrapped inside a `div` element, and then curly braces are used to include a JSX expression. Inside this JSX expression is a single `&&` operator, which is surrounded by some code both on its left and on its right sides; on the left side, the `val` value is provided, and on the right, a piece of JSX is provided.

To understand what will be output on screen, consider the following example in standard JavaScript:

[1](#)

```
true && console.log('This will show')
```

If you ran this command in the browser's console, the text 'This will show' will be output. On the flip side, consider the following example:

[1](#)

```
false && console.log('This will never show')
```

If you ran *this* command, the output will just be the boolean value of `false`. In other words, if a prop gets evaluated to `true`, using the `&&` operator, you can render whatever JSX elements you want to the right of the `&&` operator.

Completed

Conditional components

Have you ever visited a website that required a user account? To log in you click on a **Log in** button and once you've logged in, the **Log in** button changes to a **Log out** button.

This is often done using something called conditional rendering.

In a previous course, you've already learned about simple conditions using if and switch statements. Using these statements allows you to change the behaviour of code based on certain conditions being met.

For example, you can set a variable to a different value based on the result of a condition check.

```
1  
2  
3  
4  
5  
6  
  
let name;  
  
if (Math.random() > 0.5) {  
  
    name = "Mike"  
  
} else {  
  
    name = "Susan"  
  
}  
  
1  
2  
3
```

4

5

6

7

```
let name;

let newUser = true;

if (Math.random() > 0.5 && newUser) {

  name = "Mike"

} else {

  name = "Susan"

}
```

Conditional rendering is built on the same principle. By using conditions, you can return different child components. This is often done using the props that are passed into the parent component, but can also be done based on component state.

Let's take a look at a simple example.

Let's say you have two child components called `LoginButton` and `LogoutButton`; each displaying their corresponding button.

In the parent component, named `LogInOutButton`, you can check the props passed into the parent component and return a different child component based on the value of the props.

In this example, the props contains a property named `isLoggedIn`. When this is set to `true`, the `LogoutButton` component is returned. Otherwise, the `LoginButton` component is returned.

1

2

3

4

5

6

7

```
function LogInOutButton(props) {  
  
  const isLoggedIn = props.isLoggedIn;  
  
  if (isLoggedIn) {  
  
    return <LogoutButton />;  
  
  } else {  
  
    return <LoginButton />;  
  
}
```

Then when the `LogInOutButton` parent component is used, the prop can be passed in.

1

```
<LogInOutButton isLoggedIn={false} />
```

This is a simple example showing how you can change what is displayed based on a condition check. You will use this often when developing React applications.

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

- [React Router v6](#)
- [nav: The Navigation Section element](#)
- [Conditional \(ternary\) operator in JavaScript](#)
- [if...else](#)

Completed

Bundling assets

Earlier, you learned what assets are in React and the best practices for storing them in your project folders.

In this reading, you will learn about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You will also learn about the trade-offs inherent in using asset-heavy apps.

The app's files will likely be bundled when working with a React app. Bundling is a process that takes all the imported files in an app and joins them into a single file, referred to as a **bundle**.

Several tools can perform this bundling. Since, in this course, you have used the `create-react-app` to build various React apps, you will focus on webpack. This is because webpack is the built-in tool for the `create-react-app`.

Let's start by explaining what webpack is and why you need it.

Simply put, webpack is a module bundler.

Practically, this means that it will take various kinds of files, such as SVG and image files, CSS and SCSS files, JavaScript files, and TypeScript files, and it will bundle them together so that a browser can understand that bundle and work with it.

Why is this important?

When building websites, you could probably do without webpack since your project's structure might be straightforward: you may have a single CSS library, such as Bootstrap, loaded from a CDN (content delivery network). You might also have a single JavaScript file in your static HTML document. If that is all there is to it, you do not need to use webpack in such a scenario.

However, modern web development can get complex.

Here is an example of the first few lines of code in a single file of a React application:

1

2

3

4

5

6

7

8

9

10

11

```
import React from 'react';

import '@atlaskit/css-reset';

import styled from 'styled-components';

import './index.css';

import { ThemeProvider } from './contexts/theme';

import { DragDropContext } from 'react-beautiful-dnd';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Nav from './components/Nav';

import data from './data';

import Loading from './components>Loading';
```

The imports here are from fictional libraries and resources because the specific libraries are not necessary. All these different imports can be of various file types: .js, .svg, .css, and so on.

In turn, all the imported files might have their own imported files, and even those might have their imports.

This means that depending on other files, all of these files can create a **dependency graph**. The order in which all these files are loading is essential. That dependency graph can get so complex that it becomes almost impossible for a human to structure a complex project and bundle all those dependencies properly.

This is the reason you need tools like webpack.

So, webpack builds a dependency graph and bundles modules into one or more files that a browser can consume.

While it is doing that, it also does the following:

- It converts modern JS code - which can only be understood by modern browsers - into older versions of JavaScript so that older browsers can understand your code. This process is known as *transpiling*. For example, you can transpile ES7 code to ES5 code using webpack.

- It optimizes your code to load as quickly as possible when a user visits your web pages.
- It can process your SCSS code into the regular CSS, which browsers can understand.
- It can build source maps of the bundle's building blocks
- It can produce various kinds of files based on rules and templates. This includes HTML files, among others.

Another significant characteristic of webpack is that it helps developers create modern web apps.

It helps you achieve this using two modes: **production** mode or **development** mode.

In **development** mode, webpack bundles your files and optimizes your bundles for updates - so that any updates to any of the files in your locally developed app are quickly re-bundled. It also builds source maps so you can inspect the original file included in the bundled code.

In **production** mode, webpack bundles your files so that they are optimized for speed. This means the files are minified and organized to take up the least amount of memory. So, they are optimized for speed because these bundles are fast to download when a user visits the website online.

Once all the source files of your app have been bundled into a single bundle file, then that single bundle file gets served to a visitor browsing the live version of your app online, and the entire app's contents get served at once.

This works great for smaller apps, but if you have a more extensive app, this approach is likely to affect your site's speed. The longer it takes for a web app to load, the more likely the visitor will leave and move on to another unrelated website. There are several ways to tackle this issue of a large bundle.

One such approach is code-splitting, a practice where a module bundler like webpack splits the single bundle file into multiple bundles, which are then loaded on an as-needed basis. With the help of code-splitting, you can **lazy load** only the parts that the visitor to the app needs to have at any given time. This approach significantly reduces the download times and allows React-powered apps to get much better speeds.

There are other ways to tackle these problems.

An example of a viable alternative is SSR (Server-side rendering).

With SSR, React components are rendered to HTML on the server, and the visitor downloads the finished HTML code. An alternative to SSR is client-side rendering, which downloads the index.html file and then lets React inject its own code into a dedicated HTML element (the **root** element in `create-react-app`). In this course, you've only worked with client-side rendering.

Sometimes, you can combine client-side rendering and server-side rendering. This approach results in what's referred to as **isomorphic apps**.

In this reading, you learned about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You also learned about the trade-offs inherent in the use of asset-heavy apps.

Completed

Solution: Displaying images

Here's the completed App.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12

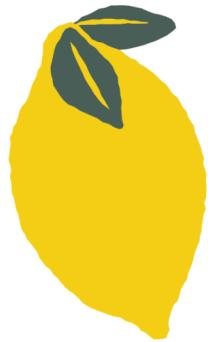
import logo from"./assets/logo.png"

function App() {
  return (
    <div className="App">
      <h1>Task: Add an image below</h1>
      <img src={logo} alt="Logo" />
    </div>
  )
}
```

```
) ;  
};  
  
export default App;
```

Here is the output from the solution code for the App.js file:

Task: Add an image below



LITTLE LEMON

Step 1

First, you imported the logo image.

1
2
3
4
5
6
7
8
9
10

```
import logo from "./assets/logo.png"

function App() {
  return (
    <div className="App">
      <h1>Task: Add an image below</h1>
    </div>
  );
}

export default App;
```

Step 2

Then, inside the `return` statement, you added the new `img` element, with the `src` attribute set to the JSX expression evaluating the `logo` value, and the `alt` attribute holding the string of "Logo".

1

2

3

4

5

6

7

8

9

10

11

12

```
import logo from"./assets/logo.png"

function App() {
  return (
    <div className="App">
      <h1>Task: Add an image below</h1>
      <img src={logo} alt="Logo" />
    </div>
  );
}

export default App;
```

Completed

Media packages

In this reading, you'll learn how to install the reactjs-media npm package.

You can find this package on the npmjs.org website at the following URL:

<https://www.npmjs.com/package/react-player>

To install this package you'll need to use the following command *in the terminal*:

1

```
npm install react-player
```

Once you have this package installed, you can start using it in your project.

There are a few ways that you can import and use the installed package. For example, to get the entire package's functionality, use the following import:

1

```
import ReactPlayer from "react-player";
```

If you are, for example, only planning to use videos from a site like YouTube, to reduce bundle size, you can use the following import:

1

```
import ReactPlayer from "react-player/youtube";
```

Here's an example of using the react-player packaged in a small React app:

1

2

3

4

5

6

7

8

9

10

```
11
12
13
14
15
16
17
18
19

import React from "react";
import ReactPlayer from "react-player/youtube";

const App = () => {
  return (
    <div>
      <MyVideo />
    </div>
  );
};

const MyVideo = () => {
```

```
return (

<ReactPlayer url='https://www.youtube.com/watch?v=ysz5S6PUM-U' />

);

};

export default App;
```

In this reading, you learned how to install and use the react-player npm package.

Completed

Solution: Song selection

Here's the completed App.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

```
import React from "react";  
  
function App() {  
  
  const bird1 = new Audio(  
  
    "https://upload.wikimedia.org/wikipedia/commons/9/9b/Hydroprogne\_caspia\_-Caspian\_Tern\_XC432679.mp3"  
  );  
  
  const bird2 = new Audio(  
  
    "https://upload.wikimedia.org/wikipedia/commons/b/b5/Hydroprogne\_caspia\_-Caspian\_Tern\_XC432881.mp3"  
  );  
}  
;
```

```
function toggle1() {  
  
  if (bird1.paused) {  
  
    bird1.play();  
  
  } else {  
  
    bird1.pause();  
  
  }  
  
};  
  
  
  
  
function toggle2() {  
  
  if (bird2.paused) {  
  
    bird2.play();  
  
  } else {  
  
    bird2.pause();  
  
  }  
  
};  
  
  
  
  
return (  
  
<div>  
  
  <button onClick={toggle1}>Caspian Tern 1</button>  
  
  <button onClick={toggle2}>Caspian Tern 2</button>  
  
</div>
```

```
) ;  
}  
  
export default App;
```

Here is the output from the solution code for the App.js file:

Caspian Tern 1 Caspian Tern 2

Step 1

In this ungraded lab, your goal was to read through the existing code of the App.js file, and update the second button so that it's running the `toggle2` function on a click to the second button.

1

2

3

4

5

6

7

8

9

10

11

12

```
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
  
import React from "react";  
  
function App() {
```

```
const bird1 = new Audio(  
  
"https://upload.wikimedia.org/wikipedia/commons/9/9b/Hydroprogne\_caspia\_-\_Caspian\_Tern\_XC432679.mp3"  
);  
  
// const bird2 = new Audio(  
  
//  
"https://upload.wikimedia.org/wikipedia/commons/b/b5/Hydroprogne\_caspia\_-\_Caspian\_Tern\_XC432881.mp3"  
// );  
  
function toggle1() {  
  
    if (bird1.paused) {  
  
        bird1.play();  
  
    } else {  
  
        bird1.pause();  
  
    }  
}  
  
};  
  
return (  
  
<div>
```

```
<button onClick={toggle1}>Caspian Tern 1</button>

<button onClick={toggle2}>Caspian Tern 2</button>

</div>

);

}

export default App;
```

Step 2

After adding the `toggle2` function to the JSX expression in the second button's `onClick` event-handling attribute, you should have un-commented the `bird2` variable on lines 9 to 11.

1

2

3

4

5

6

7

8

9

10

11

```
import React from "react";
```

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

```
function App() {  
  
  const bird1 = new Audio(  
  
    "https://upload.wikimedia.org/wikipedia/commons/9/9b/Hydroprogne_caspia_-  
    Caspian_Tern_XC432679.mp3"  
  
  );  
  
  const bird2 = new Audio(  
  
    "https://upload.wikimedia.org/wikipedia/commons/b/b5/Hydroprogne_caspia_-  
    Caspian_Tern_XC432881.mp3"  
  
  );  
  
  function toggle1() {  
  
    if (bird1.paused) {  
  
      bird1.play();  
  
    } else {  
  
      bird1.pause();  
  
    }  
  
  };  
  
  return (  
}
```

```
<div>

  <button onClick={toggle1}>Caspian Tern 1</button>

  <button onClick={toggle2}>Caspian Tern 2</button>

</div>

);

}

export default App;
```

Step 3

Next, you needed to define the `toggle2` function: it should have had the exact same functionality as the `toggle1` function, but it needed to work with the `bird2` variable (instead of the `bird1` variable as it did in the `toggle1` function).

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

```
import React from "react";

function App() {
    const bird1 = new Audio(
        "https://upload.wikimedia.org/wikipedia/commons/9/9b/Hydroprogne_caspia_-_Caspian_Tern_XC432679.mp3"
    );

    const bird2 = new Audio(
        "https://upload.wikimedia.org/wikipedia/commons/b/b5/Hydroprogne_caspia_-_Caspian_Tern_XC432881.mp3"
    );

    function toggle1() {
```

```
if (bird1.paused) {  
    bird1.play();  
}  
else {  
    bird1.pause();  
}  
};  
  
function toggle2() {  
    if (bird2.paused) {  
        bird2.play();  
    } else {  
        bird2.pause();  
    }  
};  
  
return (  
    <div>  
        <button onClick={toggle1}>Caspian Tern 1</button>  
        <button onClick={toggle2}>Caspian Tern 2</button>  
    </div>  
);
```

}

```
export default App;
```

Completed

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

- [webpack docs](#)
- [webpack asset management](#)
- [npm docs](#)
- [ReactPlayer on npm](#)
- [Video and audio content on the web](#)

Completed

week4-

About this graded assessment: Calculator app

The purpose of this graded assessment

The primary purpose of a graded assessment is to check your knowledge and understanding of the key learning objectives of the course you have just completed. Most importantly, graded assessments help you establish which topics you have mastered and which require further focus before completing the course. Ultimately, the graded assessment is designed to help you make sure that you can apply what you have learned. This assessment's learning objective is to allow you to create a React application or App.

Prepare for this graded assessment

You will have already encountered exercises, knowledge checks, in-video questions and other assessments as you have progressed through the course. The 'styling a page' ungraded lab from Module 2 is the foundation for this assessment.

The graded assessment requires you to complete a calculator in React. You will be provided with code snippets, and your task is to use these, plus any of your code to complete the calculator that can perform the four basic mathematical operations: addition, subtraction, multiplication, and division.

It will also have a single input button, which will accept user input (any number) and a total starting with a zero.

Once a user types into the input field, they will then have to update the total by pressing any of the four math operation buttons:

- addition
- subtraction
- multiplication
- division

Here's a diagram of the completed calculator app:

Simplest Working Calculator

8760

24

add subtract multiply divide reset input reset result

Nothing in the graded assessment will be outside what you have covered already, so you should be well placed to succeed.

Review the graded assessment

You will review your page to assess whether it meets the requirements outlined in the self-review quiz.

Good luck!

Completed

Solution: Build a calculator app

Here is the completed App.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

```
import React, { useState, useRef } from "react";
import "./App.css";

function App() {
```

```
const inputRef = useRef(null);

const resultRef = useRef(null);

const [result, setResult] = useState(0);

function plus(e) {
  e.preventDefault();

  // const inputVal = inputRef.current.value;
  // const newResult = result + Number(inputVal);
  // setResult(newResult);

  setResult((result) => result + Number(inputRef.current.value));
}

function minus(e) {
  e.preventDefault();
  const inputVal = inputRef.current.value;
  const newResult = result - Number(inputVal);
  setResult(newResult);
}

function times(e) {
  e.preventDefault();
```

```
const inputVal = inputRef.current.value;

const newResult = result * Number(inputVal);

setResult(newResult);

}

function divide(e) {
  e.preventDefault();

  const inputVal = inputRef.current.value;

  const newResult = result / Number(inputVal);

  setResult(newResult);

}

function resetInput(e) {
  e.preventDefault();

  inputRef.current.value = 0;
}

Here is the completed App.css file:
  1

  2

  3

  4

  5
```

6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

* {

font-family: sans-serif;

```
}

input,

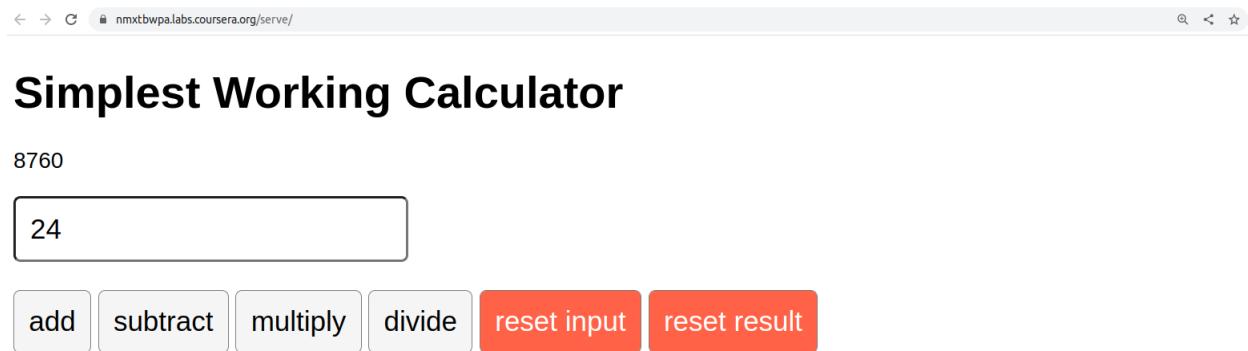
button {
    font-size: 20px;
    padding: 10px;
    border-radius: 5px;
}

input {
    display: block;
    margin-bottom: 20px;
}

button {
    border: 1px solid gray;
    background: whitesmoke;
    margin-right: 5px;
}

button:nth-last-child(2),
button:nth-last-child(1) {
    background: tomato;
    color: white;
}
```

The screenshot of the completed calculator app:



Completed

Next steps

Well done on completing this course! So far you have built up a fundamental understanding of how React works and how it uses components as the building blocks of apps. And now you're one step closer to your goal of becoming a developer. That's something to be proud of! But what might your next steps be? If you choose to continue your journey with React, you might want to take another course that will build on the React knowledge and skills you gained in this one. Or, you can expand your knowledge base and take a course on a related development topic. Just like in this course, you will do a project for the other courses too. This means that you will have another completed project for your growing portfolio which you can use to attract potential employers. That's a smart strategy! Again, congratulations on completing the course. You've built up some good momentum so keep at it!

Completed

Course6-Advanced React

Course syllabus

By the end of this reading, you will have learned about the scope of things you will cover in this course.

Prerequisites

To take this course, you should understand the basics of React, HTML, CSS, and JavaScript. Additionally, it always helps to have a can-do attitude!

Course content

This course covers advanced React development. You'll learn how to use more advanced React concepts and features, optimize and test your React applications, and become proficient in using React and JSX.

This course consists of four modules. They cover the following topics:

Module 1: Components

In this introductory module, you'll learn about React and your career opportunities. You'll also learn how to set up your coding environment so that you have as productive a learning experience as possible. The purpose of this module is to understand the what and the why behind learning React. You'll also learn about career opportunities and how to set up your coding environment for the most efficient and productive learning experience.

In React, everything revolves around components. You'll learn how to efficiently render list and form components, as well as how to lift up a shared state when several components need access to the data.

By the end of this module you will be able to:

- Outline React and various career opportunities.
- Render and transform lists with keys in React.
- Distinguish between controlled and uncontrolled React components.
- Create a controlled form component in React.
- Share component state by lifting state up to the closest common ancestor.
- Share global state using React Context.

Module 2: React Hooks and Custom Hooks

The second module of this course covers React hooks and custom hooks. You'll learn how to use all the common hooks in React, and how to put them to use within your application. You will also test your skills by building your own custom hooks.

By the end of this module you will be able to:

- Explain the uses and purpose of React hooks.
- Detail the concept and nature of state and state change.
- Use the State hook to declare, read and update the state of a component.
- Use the Effect hook to perform side-effects within a React component.
- Use hooks to fetch data in React.
- Create appropriate custom hooks in React.

Module 3: JSX and Testing

In this module, you'll learn about JSX and testing in React. You'll cover JSX in-depth and discover advanced patterns to encapsulate common behavior via Higher-order components and Render Props. You will then learn how to use Jest and the React Testing Library to write and perform tests on your applications.

By the end of this module, you will be able to:

- Define the types of children within JSX.
- Describe the process and purpose of creating render props.
- Describe the process and purpose of creating higher-order components.
- Use Jest and the React Testing Library to write and perform tests on your applications.

Module 4: Graded Assessment

In this module, you will be assessed on the key skills covered in the course and create a project to add to your portfolio.

You will be provided with code snippets, and your task will be to use these, plus any of your own code to complete your developers' portfolio.

This is a creative project, and the goal is to use as many React concepts as possible within this portfolio. You can use component composition, code reusability, hooks, manage state, interact with an external API, create forms, lists and so on.

By the end of this module, you will be able to:

- Synthesize the skills from this course to create a portfolio.
- Reflect on this course's content and on the learning path that lies ahead.

Completed

How to be successful in this course

Taking an online course can be overwhelming. How do you learn at your own pace and successfully achieve your goals?

Here are some general tips that can help you stay focused and on track.

Set daily goals for studying

Ask yourself what you hope to accomplish in your course each day. Setting a clear goal can help you stay motivated and beat procrastination. The goal should be specific and easy to measure, such as

"I'll watch all the videos in Module 2 and complete the first programming assignment". And don't forget to reward yourself when you make progress towards your goal!

Create a dedicated study space

It's easier to recall information if you're in the same place where you first learned it, so having a dedicated space at home to take online courses can make your learning more effective. Remove any distractions from the space and if possible, make it separate from your bed or sofa. A clear distinction between where you study and where you take breaks can help you focus.

Schedule time to study on your calendar

Open your calendar and choose a predictable, reliable time that you can dedicate to watching lectures and completing assignments. This helps ensure that your courses won't become the last thing on your to-do list.

Tip: You can add deadlines for a Coursera course to your Google calendar, Apple calendar, or another calendar app.

Keep yourself accountable

Tell your friends about the courses you're taking, post achievements to your social media accounts or blog about your homework assignments. Having a community and support network of friends and family to cheer you on makes a difference!

Actively take notes

Taking notes can promote active thinking, boost comprehension and extend your attention span. It's a good strategy to internalize knowledge whether you're learning online or in the classroom. So, grab a notebook or find a digital app that works best for you and start synthesizing key points.

Tip: While watching a lecture on Coursera, you can click the 'Save Note' button below the video to save a screenshot to your course notes and add your own comments.

Join the discussion

Course discussion forums are a great place to ask questions about assignments, discuss topics, share resources and make friends. Our research shows that learners who participate in the discussion forums are 37% more likely to complete a course. So make a post today!

Do one thing at a time

Multitasking is less productive than focusing on a single task at a time. Researchers from Stanford University found that "People who are regularly bombarded with several streams of electronic information cannot pay attention, recall information or switch from one job to another as well as those who complete one task at a time." Stay focused on one thing at a time. You'll absorb more

information and complete assignments with greater productivity and ease than if you were trying to do many things at once.

Take breaks

Resting your brain after learning is critical to high performance. If you find yourself working on a challenging problem without much progress for an hour, take a break. Walking outside, taking a shower or talking with a friend can help you to re-energize and even give you new ideas on how to tackle the project.

Your learning journey starts now!

While preparing for the module quiz or working on achieving your learning goals you're encouraged to:

- Work through each lesson in the learning pathway. Try not to skip any activities or lessons unless you are certain that you already know this information well enough to move ahead.
- Take the opportunity to go back and watch a video or read all the information provided before moving on to the next lesson or module.
- Complete all the knowledge and module quizzes and exercises.
- Read the feedback carefully when answering quizzes, as this will help you to reinforce what you are learning.
- Make use of the practical learning environment provided by the exercises. You can gain substantial reinforcement of your learning through the step-by-step application of your skills.

Completed

Setting up a React project in VS Code (Optional)

To complete the exercises in this course you have been provided with a dedicated lab environment set up specifically for you to apply the skills that you have learned. You can find out more about Working with Labs in this course in the following reading.

You can also use VS Code to practice these exercises on your local machine as an alternative option.

To follow along in this reading, you need to have Node.js and VS Code already installed on your computer. If you don't have this setup, please refer to the following resources in this course: Setting up VS Code and Installing Node and NPM.

In VS Code, you're ready to start a brand new React project.

You can do it using npm.

What is npm?

When Node.js is installed on a computer, **npm** comes bundled with it.

With **npm**, you can:

1. Author your own Node.js modules ("packages"), and publish them on the npm website so that other people can download and use them
2. Use other people's authored modules ("packages")

So, ultimately, npm is all about *code sharing and reuse*. You can **use other people's code** in your own projects, and you can also **publish your own Node.js modules** so that other people can use them.

An example npm module that can be useful for a new React developer is [create-react-app](#). While this npm module comes with its own website, you can also find some info on the [create-react-app project on GitHub](#).

Whenever you run the npm command to add other people's code, **that code, and all other Node modules that depend on it, get downloaded** to your machine.

However, although it's possible to do so, this is not really necessary, at least in the case of the [create-react-app](#) Node module.

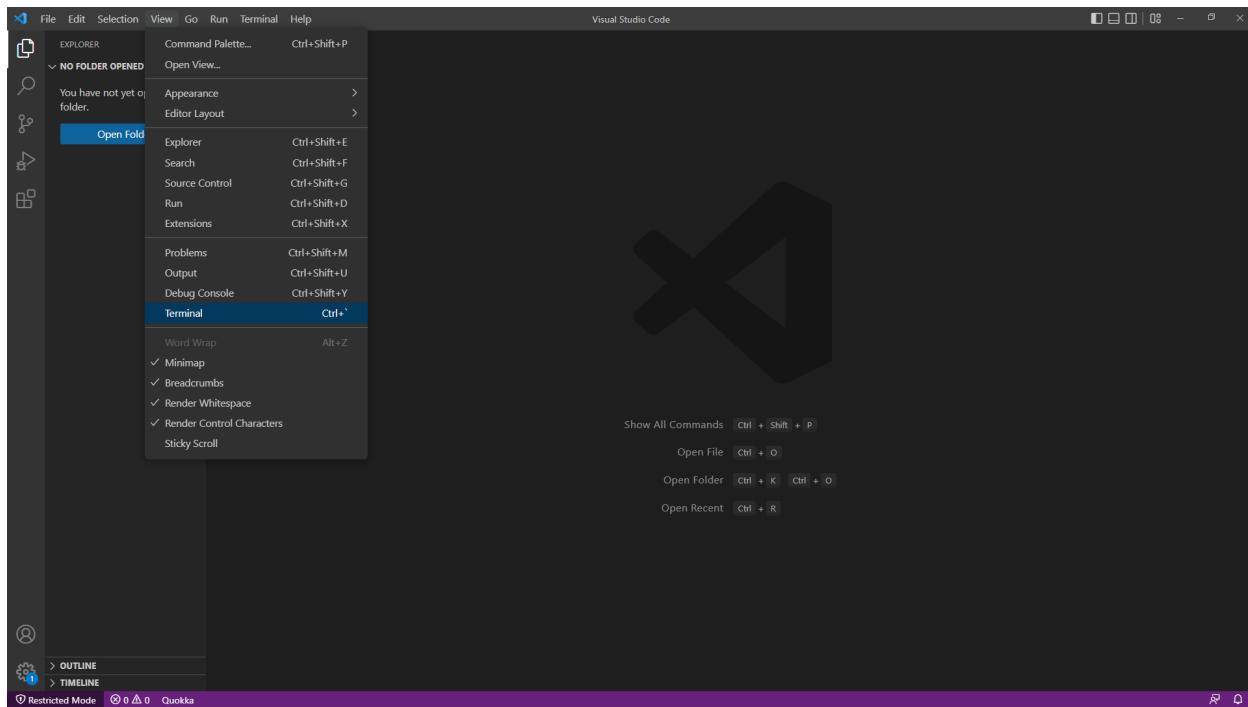
In other words, you can avoid installing the [create-react-app](#) package but still use it.

You can do that by running the following command: `npm init react-app example`, where "example" is the actual name of your app. You can use any name you'd like, but it's always good to have a name that is descriptive and short.

In the next section, you'll learn how to build a brand new app that you can name: `firstapp`.

Opening the built-in VS Code terminal and running *npm init react-app* command

In VS Code, click on *View*, *Terminal* to open the built-in terminal.



Now run the command to add a brand new React app to the machine:

1

```
npm init react-app firstapp
```

The installation and setup might take a few minutes.

Here's the output of executing the above command:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

```
Creating a new React app in /home/pc/Desktop/firstapp.
```

```
Installing packages. This might take a couple of minutes.
```

```
Installing react, react-dom, and react-scripts with cra-template...
```

```
added 1383 packages in 56s
```

```
190 packages are looking for funding
```

```
      run `npm fund` for details
```

```
Initialized a git repository.
```

```
Installing template dependencies using npm...
```

```
npm WARN deprecated source-map-resolve@0.6.0:
```

```
See https://github.com/lydell/source-map-resolve#deprecated
```

```
added 39 packages in 6s
```

```
190 packages are looking for funding
```

```
      run `npm fund` for details
```

```
Removing template package using npm...
```

```
removed 1 package, and audited 1422 packages in 3s
```

```
190 packages are looking for funding
```

```
    run `npm fund` for details
```

```
6 high severity vulnerabilities
```

```
To address all issues (including breaking changes), run:
```

```
npm audit fix --force
```

```
Run `npm audit` for details.
```

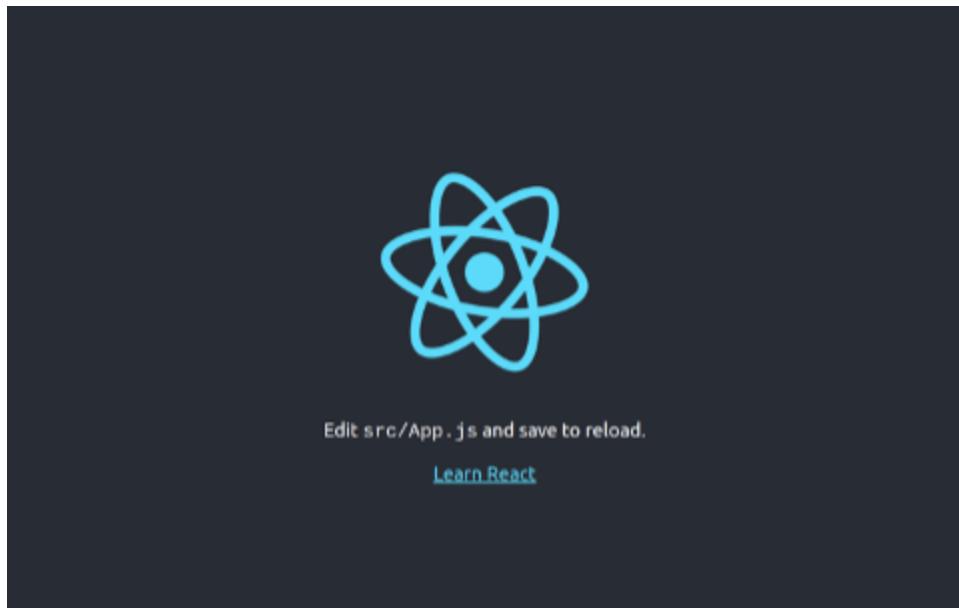
```
Created git commit.
```

```
Success! Created firstapp at /home/pc/Desktop/firstapp
```

```
Inside that directory, you can run several commands:
```

```
If you follow the suggestions from the above output, you'll run: cd firstapp, and then npm start.
```

Following the instructions, opening a browser with the address bar pointing to <http://localhost:3000>, will show the following page in your browser:



This means that you've successfully:

- Set up your local development environment
- Run the create-react-app npm package (without installing it!)
- Built a starter React app on your local machine
- Served that starter React app in your browser

After you've built your starting setup, in Module 2 you'll start working with the basic building blocks of React: components.

Completed

Installing Node and NPM (Optional)

Before installing Node.js and npm on your machine, you first need to verify if it's already installed.

Verifying the existing installation on Windows

On Windows, you can use the **WINKEY+r** shortcut key, which opens the Run window. Inside the **Open:** input of the Run window, type **cmd** and press the enter key. This will open the command prompt.

Inside the command prompt, type:

- **node --version**

If there is Node.js installed on your Windows OS, it will return a value similar to this:

- **v16.14.2**

Then you can confirm that you have npm as well, running this:

- `npm --version`

If npm is installed, you'll get output similar to this:

- `8.5.0`

Verifying the existing Node.js and npm installation on Ubuntu (Linux)

You can quickly open a new bash (terminal) window on Ubuntu by pressing the **CTRL+ALT+t** shortcut key.

In the bash window that opens, type:

- `node --version && npm --version`

Both version numbers should appear in the bash window.

Installing Node.js and npm

On Windows OS

In case Node.js and npm are not installed on your Windows OS, navigate to <https://nodejs.org>.

Locate the big download button, listing the LTS version. As of May 2022, the LTS version available for download is 16.15.0.

On Mac OS - XCode

To install brew, you need to install Xcode first. Homebrew does not come with its own compiler and it needs Xcode installed for it to work correctly. To install Xcode do the following:

1. Open a terminal.
2. Run the following:
`shell xcode-select --install`
3. A popup will appear asking you to confirm the installation. Click on the Install button.
4. Agree to the license agreement.

brew

Macs do not come with package managers like most Linux distributions. To make up for this an external tool called brew was created. To install brew, go to the official website (<https://brew.sh/>) and copy the command provided, open a terminal and run the command :

1

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once Brew is installed you can run the following command in the terminal

```
brew install node
```

Homebrew will download and install the dependencies, once this is complete, confirm the installation by typing

```
node -v
```

This will display the Node.js version

Type :

```
npm -v
```

to display the NPM version number.

On Ubuntu

Use the **CTRL+ALT+t** shortcut key to open a new bash window, then run the following commands:

- `sudo apt update`
- `sudo apt install nodejs`

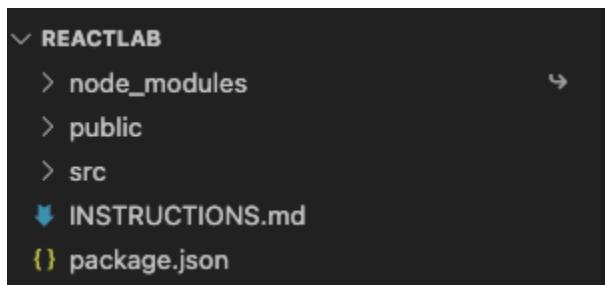
That's it, you should be all set.

For a more advanced setup and troubleshooting, please refer to the additional reading.

Completed

Working with Labs in this course

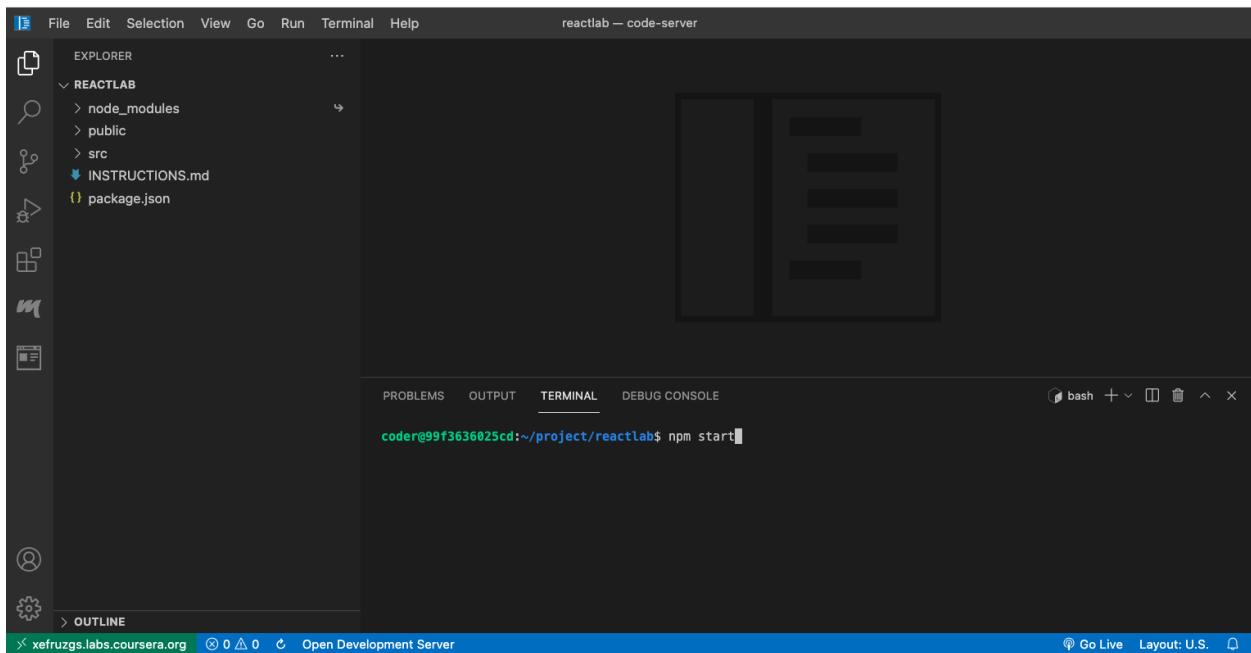
The labs for this course already have everything installed and set up so you can start working with React right away.



In order to run and view your React app you will need to open the VS Code built-in terminal, run **npm start**, and then click **Open Development server**.

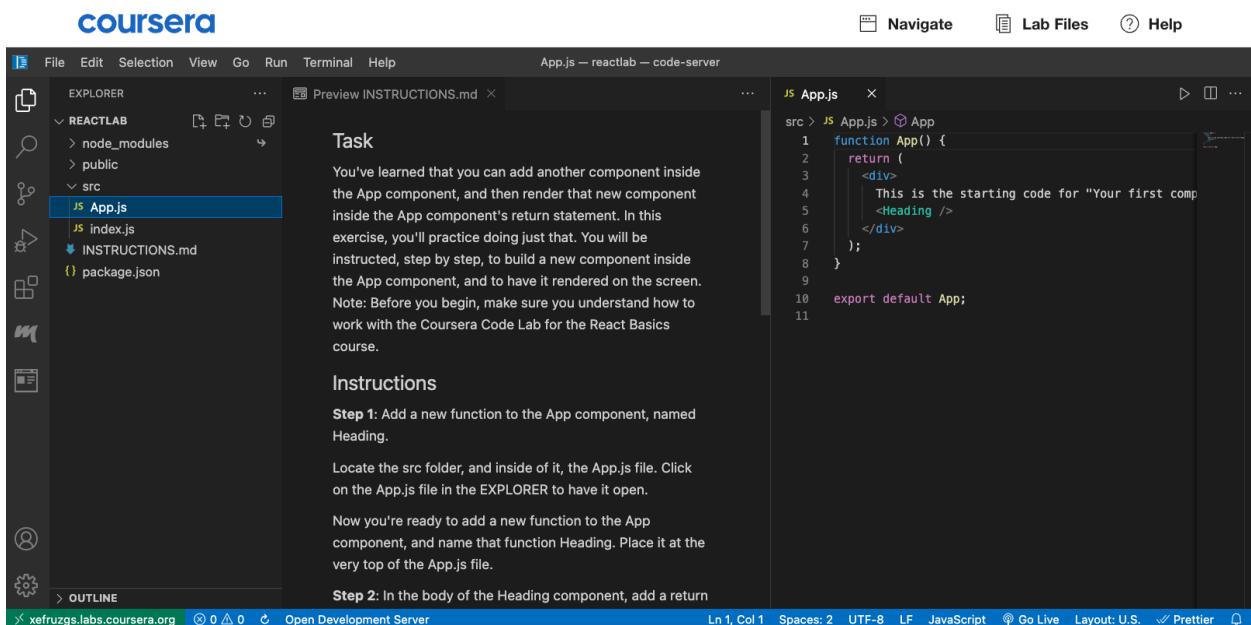
The **Open Development server** link can be found on the blue horizontal bar at the very bottom of the lab window, written in white letters on a blue background. Click this link.

That will open the app in the browser in a separate tab.

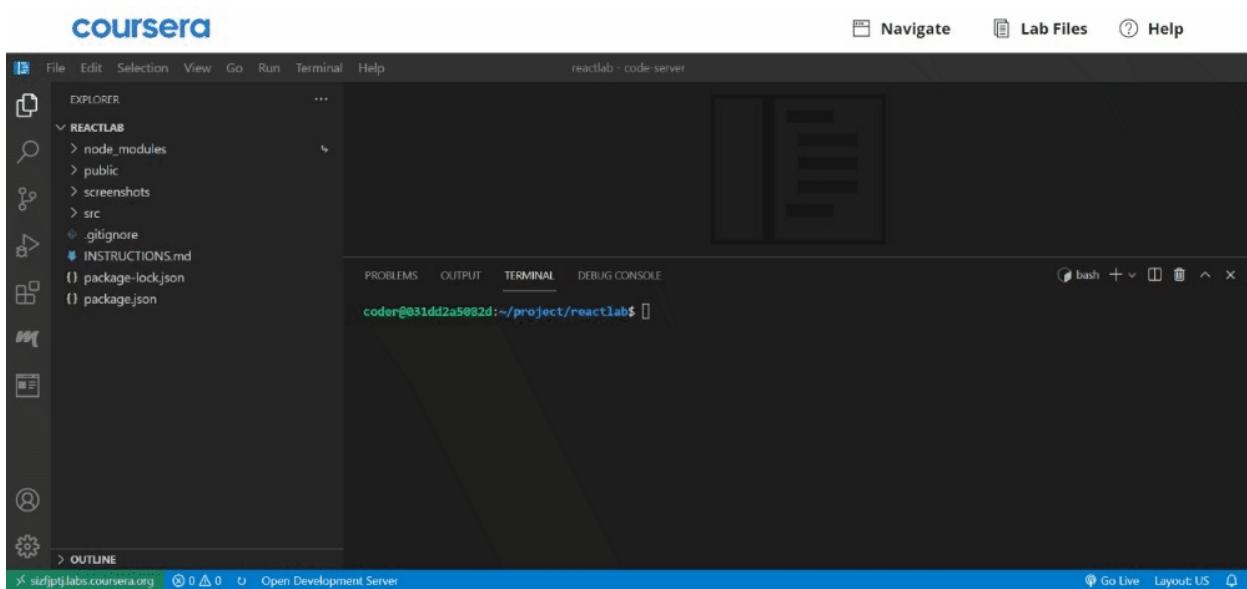


To view your code and instructions side-by-side, select the following in your VS Code toolbar:

1. View -> Editor Layout -> Two Columns
2. To view a file in Preview mode, right click on the file and open **Preview** (in the EXPLORER sidebar)
3. Select your code file in the code tree, which will open it up in a new VS Code tab.
4. You can drag any file over to the second column to view the contents in that column.
5. Great work! You can now see instructions and code at the same time.



You can download the source code in the following way (without `node_modules`):



- Click on the Lab Files tab
- Next, select `home/coder/project/reactlab`
- And click the Name checkbox to highlight all components in the project
- Then, uncheck the `node_modules` folder
- Finally, click Download to download the project

Completed

Solution: Create a basic List component

There are three types of operations you need to apply to the list of desserts: filtering, sorting and mapping.

Although the order of the operations is not that important, it's recommended to leave the final projection (mapping) to the end, since that final projection may skip some of the data needed for the filtering and sorting criteria.

Filtering

The first requirement is to display desserts that have less than 500 calories. That means Cheesecake, which has 600 cal, should be omitted. When you need to eliminate elements from your lists based on a certain condition or set of conditions, you need to use the `filter()` method.

The `filter` method creates a copy of the array, filtered down to just the elements from the original array that pass the test. In other words, it will return a new list with just the elements that fulfil the condition.

Each dessert from the list has a property called `calories`, which is an integer representing the number of calories. Therefore, the condition to be implemented should be as follows:

1

2

3

4

```
const lowCaloriesDesserts = props.data

.filter((dessert) => {

    return dessert.calories < 500;

})
```

`lowCaloriesDessert` variable will then hold a list of three desserts, without Cheesecake.

Sorting

The second requirement you have to implement is sorting the list by calories, from low to high or in ascending order. For that, arrays in JavaScript offer the `sort()` method, which sorts the elements of an array based on a comparison function provided. The `return` value from that comparison function determines how the sorting is performed:

<code>compareFn(a, b)</code> return value	sort order
<code>> 0</code>	sort <code>a</code> after <code>b</code>
<code>< 0</code>	sort <code>a</code> before <code>b</code>
<code>== 0</code>	keep original order of <code>a</code> and <code>b</code>

You can chain one operation after another. Recall that `filter` returns the new array with the filtered down elements, so `sort` can be chained right after that, as below:

1

2

3

4

5

6

7

```
const lowCaloriesDesserts = props.data
```

```
.filter((dessert) => {  
    return dessert.calories < 500;  
})  
  
.sort((a, b) => {  
    return a.calories - b.calories;  
})
```

The compare function makes sure the sorting occurs in ascending order, according to the table above.

Mapping

Finally, to apply the desired projection and display the information as requested, you can chain the `map` operator at the end and return a <1i> item with the dessert name and its calories, both separated by a dash character, and the word “cal” at the end.

The final code should look like below:

1
2
3
4
5
6
7
8
9
10
11
12

13

14

```
const lowCaloriesDesserts = props.data

.filter((dessert) => {

    return dessert.calories < 500;

})

.sort((a, b) => {

    return a.calories - b.calories;

})

.map((dessert) => {

    return (
        <li>
            {dessert.name} - {dessert.calories} cal
        </li>
    );
});
```

And the full implementation of the `DessertsList` component:

1

2

3

4

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

const DessertsList = (props) => {

  const lowCaloriesDesserts = props.data

    .filter((dessert) => {

      return dessert.calories < 500;
```

```
        }

        .sort((a, b) => {

            return a.calories - b.calories;
        })

        .map((dessert) => {

            return (
                <li>
                    {dessert.name} - {dessert.calories} cal
                </li>
            );
        });

        return <ul>{lowCaloriesDesserts}</ul>;
    }

}

export default DessertsList;
```

Final result

This is what should be displayed in your browser:

List of low calorie desserts:

- Ice Cream - 200 cal
- Tiramisu - 300 cal
- Chocolate Cake - 400 cal

Completed

Additional resources

Here is a list of additional resources for Rendering Lists in React:

- [Map\(\)](#) allows you to create new arrays populated with the results of calling a transformation function on every element.
- [Rendering lists on official React docs website](#) dives deeper into how to display multiple similar components from a collection of data, providing examples of both filtering and transformations.
- [React keys on official docs](#) offers a comprehensive set of memotecnic rules to reinforce how to use keys properly.

Completed

Controlled components vs. Uncontrolled components

This reading will teach you how to work with uncontrolled inputs in React and the advantages of controlled inputs via state design. You will also learn when to choose controlled or uncontrolled inputs and the features each option supports.

Introduction

In most cases, React recommends using controlled components to implement forms. While this approach aligns with the React declarative model, uncontrolled form fields are still a valid option and have their merit. Let's break them down to see the differences between the two approaches and when you should use each method.

Uncontrolled Inputs

Uncontrolled inputs are like standard HTML form inputs:

```
1  
2  
3  
4  
5  
6  
7  
  
const Form = () => {  
  
  return (  
    <div>  
      <input type="text" />  
    </div>  
  );  
};
```

They remember exactly what you typed, being the DOM itself that maintains that internal state. How can you then get their value? The answer is by using a React ref.

In the code below, you can see how a ref is used to access the value of the input whenever the form is submitted.

1

2

```
3
4
5
6
7
8
9
10
11
12
13

const Form = () => {

  const inputRef = useRef(null);

  const handleSubmit = () => {
    const inputValue = inputRef.current.value;
    // Do something with the value
  }

  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef} type="text" />
    </form>
  );
}
```

```
</form>

);

}
```

In other words, you must **pull** the value from the field when needed.

Uncontrolled components are the simplest way to implement form inputs. There are certainly valued cases for them, especially when your form is straightforward. Unfortunately, they are not as powerful as their counterpart, so let's look at controlled inputs next.

Controlled Inputs

Controlled inputs accept their current value as a prop and a callback to change that value. That implies that the value of the input has to live in the React state somewhere. Typically, the component that renders the input (like a form component) saves that in its state:

1
2
3
4
5
6
7
8
9
10
11
12
13

14

15

16

17

```
const Form = () => {

  const [value, setValue] = useState("");

  const handleChange = (e) => {
    setValue(e.target.value)
  }

  return (
    <form>
      <input
        value={value}
        onChange={handleChange}
        type="text"
      />
    </form>
  );
}
```

Every time you type a new character, the `handleChange` function is executed. It receives the new value of the input, and then it sets it in the state. In the code example above, the flow would be as follows:

- The input starts out with an empty string: `""`
- You type “a” and `handleChange` gets an “a” attached in the event object, as `e.target.value`, and subsequently calls `setValue` with it. The input is then updated to have the value of “a”.
- You type “b” and `handleChange` gets called with `e.target.value` being “ab”.and sets that to the state. That gets set into the state. The input is then re-rendered once more, now with `value = "ab"` .

This flow **pushes** the value changes to the form component instead of pulling like the ref example from the uncontrolled version. Therefore, the Form component always has the input's current value without needing to ask for it explicitly.

As a result, your data (React state) and UI (input tags) are always in sync. Another implication is that forms can respond to input changes immediately, for example, by:

- Instant validation per field
- Disabling the submit button unless all fields have valid data
- Enforcing a specific input format, like phone or credit card numbers

Sometimes you will find yourself not needing any of that. In that case uncontrolled could be a more straightforward choice.

The file input type

There are some specific form inputs that are always uncontrolled, like the file input tag.

In React, an `<input type="file" />` is always an uncontrolled component because its value is read-only and can't be set programmatically.

The following example illustrates how to create a ref to the DOM node to access any files selected in the form submit handler:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

```
const Form = () => {

  const fileInput = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();

    const files = fileInput.current.files;

    // Do something with the files here

  }

  return (
    <form onSubmit={handleSubmit}>
```

```

<input
  ref={fileInput}
  type="file"
/>

</form>

);
}

;

```

Conclusion

Uncontrolled components with refs are fine if your form is incredibly simple regarding UI feedback. However, controlled input fields are the way to go if you need more features in your forms. Evaluate your specific situation and pick the option that works best for you. The below table summarizes the features that each one supports:

Feature	Uncontrolled	Controlled
One-time value retrieval (e.g. on submit)	Yes	Yes
Validating on submit	Yes	Yes
Instant field validation	No	Yes
Conditionally disabling a submit button	No	Yes
Enforcing a specific input format	No	Yes
Several inputs for one piece of data	No	Yes
Dynamic inputs	No	Yes

And that's it about controlled vs. uncontrolled components. You have learned in detail about each option, when to pick one or another, and finally, a comparison of the features supported.

Completed

Solution: Create a registration form

Here is the completed solution code for the App.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import './App.css';

import {useState} from "react";

import {validateEmail} from "../src/utils";



const PasswordErrorMessage = () => {

  return (

    <p className="FieldError">Password should have at least 8
    characters</p>
  );
}

function App() {

  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState({
    value: "",
    isTouched: false,
  });
  const [role, setRole] = useState("role");
}
```

```
const getIsValid = () => {

  return (
    firstName &&
    validateEmail(email) &&
    password.value.length >= 8 &&
    role !== "role"
  );
};

const clearForm = () => {
  setFirstName("");
  setLastName("");
  setEmail("");
  setPassword({
    value: "",
    isTouched: false,
  });
  setRole("role");
};


```

Step 1

The first step involves converting all form elements into controlled components. Since the pieces of local state have been already defined at the top of the component, you just have to assign each state piece to the `value` prop from each input element. To be able to account for state updates, each input should also define the `onChange` prop and call the state setter with the value property from the event target as parameter.

The password input is a special case that has an object as state instead of a string. As a result, the state setter should spread the previous values so they don't get overridden. Finally, to make sure the password characters are obscured, you need to use the type "password" for the input.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

```
<div className="Field">

  <label>
    First name <sup>*</sup>
  </label>

  <input
    value={firstName}
    onChange={(e) => {
      setFirstName(e.target.value);
    }}
    placeholder="First name"
  />

</div>

<div className="Field">
  <label>Last name</label>
  <input
    value={lastName}
    onChange={(e) => {
      setLastName(e.target.value);
    }}
  />
```

```
placeholder="Last name"

/>

</div>

<div className="Field">

<label>

  Email address <sup>*</sup>

</label>

<input

  value={email}

  onChange={(e) => {

    setEmail(e.target.value);

  }}

  placeholder="Email address"

/>

</div>

<div className="Field">

<label>

  Password <sup>*</sup>

</label>

<input

  value={password.value}
```

Step 2

The `isTouched` property on the password state was defined to determine when the input was touched at least once. In order to listen for interactions, form inputs have two additional events you can subscribe to: `onFocus` and `onBlur`.

In this scenario, you need to use the `onBlur` event, which is called whenever the input loses focus, so that guarantees the user has interacted with the password input at least once. In that event, you should set the `isTouched` property to `true` with the password state setter.

Then, the condition to display the error message relies on that value being `true` and a check on the password length to see if it's less than 8 characters long. If the condition is `true`, the component `PasswordErrorMessage` should be rendered. The final code should be as follows:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

17

18

19

20

```
<div className="Field">

  <label>

    Password <sup>*</sup>

  </label>

  <input

    value={password.value}

    type="password"

    onChange={(e) => {

      setPassword({ ...password, value: e.target.value });

    }}

    onBlur={() => {

      setPassword({ ...password, isTouched: true });

    }}

    placeholder="Password"

  />

  {password.isTouched && password.value.length < 8 ? (
    <PasswordErrorMessage />
  )}
```

```
) : null}  
</div>
```

If implemented correctly, the form should display an error message below the password field:

Password *

.....

>Password should have at least 8 characters

Role *

Role

Step 3

To prevent the default behavior of the form when clicking on the submit button, you have to call `preventDefault` on the event object, right in your submit handler function.

- 1
- 2
- 3
- 4
- 5

```
const handleSubmit = (e) => {  
  
  e.preventDefault();  
  
  alert("Account created!");  
  
  clearForm();  
  
};
```

Step 4

To fulfil the validation rules of the form, the body of the `getisValid` function should be implemented as below:

```
1
2
3
4
5
6
7
8

const getisValid = () => {

  return (
    firstName &&
    validateEmail(email) &&
    password.value.length >= 8 &&
    role !== "role"
  );
}
```

Below is an example of a valid form:

Sign Up

First name *

Last name

Email address *

Password *

Role *

CREATE ACCOUNT

Step 5

Finally, to clear the form state after a successful submission, you should set each piece of state to its initial value:

[1](#)

[2](#)

[3](#)

```
4  
5  
6  
7  
8  
9  
10  
  
const clearForm = () => {  
  
  setFirstName("");  
  
  setLastName("");  
  
  setEmail("");  
  
  setPassword({  
  
    value: "",  
  
    isTouched: false,  
  
  });  
  
  setRole("role");  
  
};
```

Now, when you submit the form, an alert is displayed and right after you dismiss it, the form is reset to its initial values.

Completed

Additional resources

Here is a list of additional resources for Module 1, Lesson 3 (Forms in React):

- [Forms from the official React docs](#) illustrate some examples of how React deals with certain form fields compared to traditional HTML tags, like the text area, select and file input tags. It also showcases how to handle multiple inputs by leveraging `event.target.name` and the implications of using null as a value in a controlled input.
- [Formik](#) is the most popular open source form library for React. It saves you lots of time when building forms and offers a declarative, intuitive and adoptable paradigm.
- [Yup](#) is an open-source library that integrates perfectly with Formik. It allows you to set all your form validation rules declaratively.
- [React-hook-form](#) is another popular library to easily manage your form state and validation rules.

Completed

Solution: Create a light-dark theme switcher

Here is the completed solution code for the `ThemeContext.js` file:

1

2

3

4

5

6

7

8

9

10

```
1 // Context API is used here to share the theme across the application
2 import React, { createContext, useState } from 'react';
3
4 const ThemeContext = createContext();
5
6 const ThemeProvider = ({ children }) => {
7   const [theme, setTheme] = useState('light');
8
9   const toggleTheme = () => {
10     setTheme((current) => current === 'light' ? 'dark' : 'light');
11   };
12
13   return (
14     <ThemeContext.Provider value={{ theme, toggleTheme }}>
15       {children}
16     </ThemeContext.Provider>
17   );
18 };
19
20 export default ThemeProvider;
```

```
11
12
13
14
15
16
17
18
19
20
21

import { createContext, useContext, useState } from "react";

const ThemeContext = createContext(undefined);

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider
      value={{

        theme,
        setTheme
      }}
    >{children}</ThemeContext.Provider>
  );
}
```

```
theme,  
  
  toggleTheme: () => setTheme(theme === "light" ? "dark" : "light"),  
  
} }  
  
>  
  
{children}  
  
</ThemeContext.Provider>  
  
) ;  
  
};  
  
  
export const useTheme = () => useContext(ThemeContext);
```

Here is the solution code for the **Switch/index.js** file:

1
2
3
4
5
6
7
8
9

```
10
11
12
13
14
15
16
17
18
19

import "./Styles.css";

import { useTheme } from "../ThemeContext";

const Switch = () => {

  const { theme, toggleTheme } = useTheme();

  return (
    <label className="switch">
      <input
        type="checkbox"
        checked={theme === "light"}
        onChange={toggleTheme}
      />
    </label>
  );
}

export default Switch;
```

```

        />

        <span className="slider round" />

    </label>

);

};

export default Switch;

```

Steps

Step 1

To create the `ThemeProvider`, the first step is to create a new context object, `ThemeContext`, using `createContext`, a function that can be imported from React. The default value argument is only used when a component does not have a matching Provider above it in the tree. This default value can be helpful for testing components in isolation without wrapping them. For the purpose of this exercise, it's not relevant, so `undefined` can be used.

Then, inside the `ThemeProvider` component, you need to define a new piece of local state for the theme, which can be a string whose value is either `"light"` or `"dark"`. It can be initialized to `"light"`, which is usually the default theme for applications.

In the `return` statement, the `ThemeContext.Provider` component should be rendered and wrap the children.

Finally, recall that the value prop for `ThemeContext.Provider` is what gets injected down the tree as context. Since the application needs both the theme value and a way to toggle it, two values are injected: `theme` and `toggleTheme`.

`theme` is just the light-dark theme string value, whereas `toggleTheme` is a function that receives no parameters and just toggles the theme from light to dark and vice versa.

That completes the implementation of the `ThemeProvider` component, as per the code below:

1

2

3

4

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

import { createContext, useContext, useState } from "react";

const ThemeContext = createContext(undefined);
```

```

export const ThemeProvider = ({ children }) => {

  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider
      value={ {
        theme,
        toggleTheme: () => setTheme(theme === "light" ? "dark" : "light"),
      } }
    >
      {children}
    </ThemeContext.Provider>
  );
}

```

Step 2

The implementation for `useTheme` is quite simple. You just need to import the `useContext` hook from React and pass as an argument the `ThemeContext` object defined before. That allows your components to access both `theme` and `toggleTheme` values, which are the ones the `useTheme` custom hook returns.

1

```
export const useTheme = () => useContext(ThemeContext);
```

Step 3

The `Switch` component can then be connected to the `toggleTheme` function returned from `useTheme` as per the code below:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14

const Switch = () => {

  const { theme, toggleTheme } = useTheme();

  return (
    <label className="switch">
      <input
        type="checkbox"
        checked={theme === "light"} // Add this line
        onChange={toggleTheme} // Add this line
      />
    </label>
  );
}
```

```
checked={theme === "light"}  
  
onChange={toggleTheme}  
  
/>  
  
<span className="slider round" />  
  
</label>  
  
);  
  
};
```

Step 4

And, finally, you should be able to use the switch widget on the top right corner to change the theme of the application:



When it comes to dough

We are a pizza loving family. And for years, I searched and searched and searched for the perfect pizza dough recipe. I tried dozens, or more. And while some were good, none of them were that recipe that would make me stop trying all of the others.



Completed

How re-rendering works with Context

In this reading you will learn about the default behavior of React rendering and when context is used. You will discover how to prevent unnecessary top-level re-renders with `React.memo` and how object references work in JavaScript. You will also learn how to utilize the `useMemo` hook to guarantee object references don't change during re-rendering.

So far, you have learned that when a component consumes some context value and the value of this context changes, that component re-renders.

But what happens with all components in between? Is React wise enough to only re-render the consumers and bypass the intermediary components in the tree? Well, as it turns out, that doesn't always happen and extra care should be taken when designing your React Context.

When it comes to the default behavior of React rendering, if a component renders, React will recursively re-render all its children regardless of props or context. Let's illustrate this point with an example that uses some context.

Imagine the following component structure, where the top level component injects a Context provider at the top:

```
App (ContextProvider) > A > B > C
```

1

2

3

4

5

6

7

8

9

10

11

```

const App = () => {

  return (
    <AppContext.Provider>
      <ComponentA />
    </AppContext.Provider>
  );
}

const ComponentA = () => <ComponentB />;
const ComponentB = () => <ComponentC />;
const ComponentC = () => null;

```

If the outermost App component re-renders for whatever reason, all `ComponentA`, `ComponentB` and `ComponentC` components will re-render as well, following this order:

`App (ContextProvider) -> A -> B -> C`

If some of your top level components are complex in nature, this could result in some performance hit. To mitigate this issue, you can make use of the top level API `React.memo()`.

If your component renders the same result given the same props, you can wrap it in a call to `React.memo` for a performance boost by memoizing the result.

Memoization is a programming technique that accelerates performance by caching the return values of expensive function calls.

This means that React will skip rendering the component, and reuse the last rendered result. This is a trivial case for `ComponentA`, since it doesn't receive any props.

```
const ComponentA = React.memo(() => <ComponentB />);
```

`React.memo` takes the component definition as a first argument. An optional second argument can be included if you would like to specify some custom logic that defines when the component should re-render based on previous and current props.

After that little adjustment, you will prevent any rendering from happening in all `ComponentA`, `ComponentB` and `ComponentC` if the App component re-renders.

1

2

```
3
4
5
6
7
8
9
10
11

const App = () => {
  return (
    <AppContext.Provider>
      <ComponentA />
    </AppContext.Provider>
  );
}

const ComponentA = React.memo(() => <ComponentB />);

const ComponentB = () => <ComponentC />;

const ComponentC = () => null;
```

A good rule of thumb is to wrap the React component right after your context provider with `React.memo`.

In real-life applications, you will find yourself in need of passing several pieces of data as context value, rather than a single primitive like a string or number, so you'll be working most likely with JavaScript objects.

Now, according to React context rules, all consumers that are descendants of a provider will re-render whenever the provider's value prop changes.

Let's go through the following scenario built upon the previous example, where the context value that gets injected is defined as an object called `value` with two properties, 'a' and 'b', being both strings. Also, `ComponentC` is now a consumer of context, so any time the provider `value` prop changes, `ComponentC` will re-render.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
const App = () => {
```

```

const value = {a: 'hi', b: 'bye'};

return (
  <AppContext.Provider value={value}>
    <ComponentA />
  </AppContext.Provider>
);

};

const ComponentA = React.memo(() => <ComponentB />);

const ComponentB = () => <ComponentC />;

const ComponentC = () => {
  const contextValue = useContext(AppContext);

  return null;
};

```

Imagine that the value prop from the provider changes to `{a: 'hello', b: 'bye'}`.

If that happens, the sequence of re-renders would be:

`App (ContextProvider) -> C`

That's all fine and expected, but what would happen if the App component re-renders for any other reason and the provider value doesn't change at all, being still `{a: 'hi', b: 'bye'}`?

It may be a surprise to you to find out that the sequence of re-renders is the same as before:

`App (ContextProvider) -> C`

Even though the provider value doesn't seem to change, `ComponentC` gets re-rendered.

To understand what's happening, you need to remember that in JavaScript, the below assertion is true:

```
{a: 'hi', b: 'bye'} !== {a: 'hi', b: 'bye'}
```

That is because object comparison in JavaScript is done by reference. Every time a new re-render happens in the App component, a new instance of the `value` object is created, resulting in the provider performing a comparison against its previous value and determining that it has changed, hence informing all context consumers that they should re-render.

This problem can be resolved by using the `useMemo` hook from React as follows.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
const App = () => {
  const a = 'hi';
}
```

```

const b = 'bye';

const value = useMemo(() => ({a, b}), [a, b]);

return (
  <AppContext.Provider value={value}>
    <ComponentA />
  </AppContext.Provider>
);

};

const ComponentA = React.memo(() => <ComponentB />);

const ComponentB = () => <ComponentC />;

const ComponentC = () => {
  const contextValue = useContext(AppContext);

  return null;
};

```

Hooks will be covered in depth in the next module, so don't worry too much if this is new for you. For the purpose of this example, it suffices to say that `useMemo` will memoize the returned value from the function passed as the first argument and will only re-run the computation if any of the values are passed into the array as a second argument change.

With that implementation, if the App re-renders for any other reason that does not change any of 'a' or 'b' values, the sequence of re-renders will be as such:

`App (ContextProvider)`

This is the desired result, avoiding an unnecessary re-render on `ComponentC`. `useMemo` guarantees keeping the same object reference for the `value` variable and since that's assigned to the provider's value, it determines that the context has not changed and should not notify any consumer.

Conclusion

You have learned about how re-rendering works in React when context is used and how `React.memo` and `useMemo` APIs from React can help you perform some optimizations to avoid unnecessary re-renders in your components tree.

Completed

Additional resources

Here is a list of additional resources for React Context.

- [React.memo from the official React docs](#), an API that can be used in conjunction with Context Providers to prevent unnecessary re-renders in top-level components in the tree.
- [useMemo from the official React docs](#), a hook to guarantee referential equality on objects across rendering passes.

Completed

Week2-

Working with complex data in useState

In this reading, you will learn how to use objects as state variables when using `useState`. You will also discover the proper way to only update specific properties, such as state objects and why this is done. This will be demonstrated by exploring what happens when changing the string data type to an object.

An example of holding state in an object and updating it based on user-generated events

When you need to hold state in an object and update it, initially, you might try something like this:

1

2

3

4

5

```
6
7
8
9
10
11
12
13
14
15
16
17

import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);

  function updateGreeting() {
    setGreeting({ greet: "Hello, World-Wide Web" });
  }
}
```

```
return (  
  <div>  
    <h1>{greeting.greet}</h1>  
    <button onClick={updateGreeting}>Update greeting</button>  
  </div>  
) ;  
}
```

While this works, it's not the recommended way of working with state objects in React, this is because the state object usually has more than a single property, and it is costly to update the entire object just for the sake of updating only a small part of it.

The correct way to update the state object in React when using useState

The suggested approach for updating the state object in React when using `useState` is to copy the state object and then update the copy.

This usually involves using the spread operator (...).

Keeping this in mind, here's the updated code:

1
2
3
4
5
6
7
8
9

```
10
11
12
13
14
15
16
17
18
19

import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });

  console.log(greeting, setGreeting);

  function updateGreeting() {
    const newGreeting = {...greeting};
    newGreeting.greet = "Hello, World-Wide Web";
    setGreeting(newGreeting);
  }
}
```

```
return (  
  <div>  
    <h1>{greeting.greet}</h1>  
    <button onClick={updateGreeting}>Update greeting</button>  
  </div>  
) ;  
}
```

Incorrect ways of trying to update the state object

To prove that a copy of the old state object is needed to update state, let's explore what happens when you try to update the old state object directly:

1

2

3

4

5

6

7

8

9

10

11

```
12  
13  
14  
15  
16  
17  
18  
  
import { useState } from "react";  
  
export default function App() {  
  
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });  
  
  console.log(greeting, setGreeting);  
  
  function updateGreeting() {  
  
    greeting = {greet: "Hello, World-Wide Web"};  
  
    setGreeting(greeting);  
  
  }  
  
  return (  
    <div>  
      <h1>{greeting.greet}</h1>
```

```
<button onClick={updateGreeting}>Update greeting</button>

</div>

);

}

}
```

The above code does not work because it has a **TypeError** hiding inside of it.

Specifically, the **TypeError** is: "Assignment to constant variable".

In other words, you cannot reassign a variable declared using `const`, such as in the case of the `useState` hook's array destructuring:

```
1

const [greeting, setGreeting] = useState({ greet: "Hello, World" });


```

Another approach you might attempt to use to work around the suggested way of updating state when working with a state object might be the following:

```
1
2
3
4
5
6
7
8
9
10
11
12
```

13

14

15

16

17

18

```
import { useState } from "react";

export default function App() {

  const [greeting, setGreeting] = useState({ greet: "Hello, World" });

  console.log(greeting, setGreeting);

  function updateGreeting() {
    greeting.greet = "Hello, World-Wide Web";
    setGreeting(greeting);
  }

  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

```
</div>

) ;

}
```

The above code is problematic because it doesn't throw any errors; however, it also doesn't update the heading, so it is not working correctly. This means that, regardless of how many times you click the "Update greeting" button, it will still be "Hello, World".

To reiterate, the proper way of working with state when it's saved as an object is to:

1. Copy the old state object using the spread (...) operator and save it into a new variable and
2. Pass the new variable to the state-updating function

Updating the state object using arrow functions

Now, let's use a more complex object to update state.

The state object now has two properties: greet and location.

The intention of this update is to demonstrate what to do when only a specific property of the state object is changing, while keeping the remaining properties unchanged:

1
2
3
4
5
6
7
8
9
10
11
12

```
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
  
import { useState } from "react";  
  
export default function App() {  
  const [greeting, setGreeting] = useState(  
    {  
      greet: "Hello",  
      place: "World"  
    }  
  );  
}  
};
```

```

        console.log(greeting, setGreeting);

    }

    function updateGreeting() {
        setGreeting(prevState => {
            return { ...prevState, place: "World-Wide Web" }
        });
    }

    return (
        <div>
            <h1>{greeting.greet}, {greeting.place}</h1>
            <button onClick={updateGreeting}>Update greeting</button>
        </div>
    );
}

```

The reason this works is because it uses the previous state, which is named `prevState`, and this is the previous value of the `greeting` variable. In other words, it makes a copy of the `prevState` object, and updates only the `place` property on the copied object. It then returns a brand-new object:

1

```
return { ...prevState, place: "World-Wide Web" }
```

Everything is wrapped in curly braces so that this new object is built correctly, and it is returned from the call to `setGreeting`.

Conclusion

You have learned what happens when changing the string data type to an object, with examples of holding state in an object and updating it based on user-generated events. You also learned about

correct and incorrect ways to update the state object in React when using `useState`, and about updating the state object using arrow functions.

Completed

Solution: Managing state within a component

Here is the completed solution code for the `App.js` file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import { useState } from "react";

export default function App() {

  const [giftCard, setGiftCard] = useState(
    {
      firstName: "Jennifer",
      lastName: "Smith",
      text: "Free dinner for 4 guests",
      valid: true,
      instructions: "To use your coupon, click the button below.",
    }
  );

  function spendGiftCard() {
    setGiftCard(prevState => {
      return {
        ...prevState,
        text: "Your coupon has been used.",
      }
    });
  }
}
```



```
{  
  giftCard.valid && (  
    ...
```

Here is the output from the solution code for the `App.js` file.

Gift Card Page

Customer: Jennifer Smith

Your coupon has been used.

Please visit our restaurant to renew your gift card.

Step-by-step solution

Step 1

You opened the `App.js` file and located the `spendGiftCard()` function.

Inside the `spendGiftCard()` function, you invoked the `setGiftCard()` state-updating function, without passing it any parameters or doing anything else with it.

1

2

3

```
function spendGiftCard() {  
  
  setGiftCard()  
  
}
```

Step 2

Inside the `setGiftCard()` function invocation's parentheses, you passed in an arrow function.

This arrow function has a single parameter, named `prevState`. After the arrow, you added a block of code.

1

2

3

4

5

```
function spendGiftCard() {  
  setGiftCard(prevState => {  
    } )  
}  
}
```

Step 3

Next, you returned a copy of the `prevState` object using the rest operator.

1

2

3

4

5

```
function spendGiftCard() {  
  setGiftCard(prevState => {  
    return ...prevState  
  } )  
}
```

Step 4

Next, you combined this copy of the `prevState` object with those properties that you wanted updated, by updating some of the key-value pairs that already exist on the state object that were initially passed to the `useState()` function call.

1

```
2  
3  
4  
5  
6  
7  
8  
  
function spendGiftCard() {  
  
  setGiftCard(prevState => {  
  
    return {  
      ...prevState,  
  
      text: "Your coupon has been used.",  
  
    }  
  })  
  
}  
}
```

Step 5

Finally, you updated the remaining properties on the state object.

You updated the valid key's value to `false`.

Then, updated the instructions key's value to Please visit our restaurant to renew your gift card.

1
2
3
4

5

6

7

8

9

10

```
function spendGiftCard() {  
  
  setGiftCard(prevState => {  
  
    return {  
  
      ...prevState,  
  
      text: "Your coupon has been used.",  
  
      valid: false,  
  
      instructions: "Please visit our restaurant to renew your gift card.",  
  
    }  
  }) ;  
  
}  
  
Completed
```

What is the useEffect hook?

You have been introduced to the primary usage of the `useEffect` hook, a built-in React hook best suited to perform side effects in your React components.

In this reading you will be introduced to the correct usage of the dependency array and the different `useEffect` calls that can be used to separate different concerns. You will also learn how you can clean up resources and free up memory in your `useEffect` logic by returning a function.

The code you place inside the `useEffect` hook always runs after your component mounts or, in other words, after React has updated the DOM.

In addition, depending on your configuration via the dependencies array, your effects can also run when certain state variables or props change.

By default, if no second argument is provided to the `useEffect` function, the effect will run after every render.

1

2

3

```
useEffect(() => {  
  
  document.title = 'Little Lemon';  
  
});
```

However, that may cause performance issues, especially if your side effects are computationally intensive. A way to instruct React to skip applying an effect is passing an array as a second parameter to `useEffect`.

In the below example, the integer variable `version` is passed as the second parameter. That means that the effect will only be re-run if the `version` number changes between renders.

1

2

3

```
useEffect(() => {  
  
  document.title = `Little Lemon, v${version}`;  
  
}, [version]); // Only re-run the effect if version changes
```

If `version` is 2 and the component re-renders and `version` still equals 2, React will compare `[2]` from the previous render and `[2]` from the next render. Since all items inside the array are the same, React would skip running the effect.

Use multiple Effects to Separate Concerns

React doesn't limit you in the number of effects your component can have. In fact, it encourages you to group related logic together in the same effect and break up unrelated logic into different effects.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

function MenuPage(props) {
  const [data, setData] = useState([]);
  useEffect(() => {
    document.title = 'Little Lemon';
  }, []);
}
```

```
}, []);  
  
useEffect(() => {  
  
  fetch(`https://littlelemon/menu/\${id}`)  
  
  .then(response => response.json())  
  
  .then(json => setData(json));  
  
}, [props.id]);  
  
// ...  
}  
}
```

Multiple hooks allow you to split the code based on what it is doing, improving code readability and modularity.

Effects with Cleanup

Some side effects may need to clean up resources or memory that is not required anymore, avoiding any memory leaks that could slow down your applications.

For example, you may want to set up a subscription to an external data source. In that scenario, it is vital to perform a cleanup after the effect finishes its execution.

How can you achieve that? In line with the previous point of splitting the code based on what it is doing, the `useEffect` hook has been designed to keep the code for adding and removing a subscription together, since it's tightly related.

If your effect returns a function, React will run it when it's time to clean up resources and free unused memory.

1

2

3

4

5

```
6
7
8
9
10
11
12
13
14

function LittleLemonChat(props) {
  const [status, chatStatus] = useState('offline');

  useEffect(() => {
    LemonChat.subscribeToMessages(props.chatId, () => setStatus('online'))
    return () => {
      setStatus('offline');
      LemonChat.unsubscribeFromMessages(props.chatId);
    };
  }, []);
}
```

```
// ...  
}
```

Returning a function is optional and it's the mechanism React provides in case you need to perform additional cleanup in your components.

React will make sure to run the cleanup logic when it's needed. The execution will always happen when the component unmounts. However, in effects that run after every render and not just once, React will also clean up the effect from the previous render before running the new effect next time.

Conclusion

In this lesson, you learned some practical tips for using the built-in Effect hook. In particular, you were presented with how to use the dependency array properly, how to separate different concerns in different effects, and finally how to clean up unused resources by returning an optional function inside the effect.

Completed

Additional resources

Below is a list of additional resources as you continue to explore React hooks and custom hooks. In particular, to complement your learning in the “Getting started with hooks” lesson, you can work through the following:

- The article on [destructuring](#) assignment describes how the destructuring assignment, which allows you to get values out of the array that gets returned when the `useState` hook is invoked, works in more detail.
- [The read props inside the child component](#) link on the Beta version of React docs discusses how to use destructuring assignment to get values out of the props object.
- The [useState reference on official React docs website](#) helps you understand how to work with this hook and some of the caveats involved.
- The [useEffect reference on official React docs website](#) helps you understand the syntax of this hook and goes into some depth to explain how to use and troubleshoot the `useEffect` hook.

Completed

Data fetching using hooks

You learned more about fetching data using hooks and that fetching data from a third-party API is considered a side-effect that requires the use of the `useEffect` hook to deal with the Fetch API calls in React.

You also explored how the response from fetching third-party data might fail, or be delayed, and that it can be useful to provide different renders, based on whether or not the data has been received.

In this reading, you will explore the different approaches to setting up the `useEffect` hook when fetching JSON data from the web. You will also learn why it can be useful to provide different renders, based on whether or not the data has been received.

You have previously learned about using the Fetch API to get some JSON data from a third-party website in plain JavaScript.

You'll be glad to learn that data fetching is not that different in React.

There is only one more ingredient that you need to keep in mind when working with React, namely, that fetching data from a third-party API is considered a side-effect.

Being a side-effect, you need to use the `useEffect` hook to deal with using the Fetch API calls in React.

To understand what that entails, let me illustrate it with a code example where a component is fetching some data from an external API to display information about a cryptocurrency.

1

2

3

4

5

6

7

8

9

10

11

12

```
13
14
15
16
17
18
19
20
21
22

import { useState, useEffect } from "react";

export default function App() {
  const [btcData, setBtcData] = useState({});

  useEffect(() => {
    fetch(`https://api.coindesk.com/v1/bpi/currentprice.json`)
      .then((response) => response.json())
      .then((jsonData) => setBtcData(jsonData.bpi.USD))
      .catch((error) => console.log(error));
  }, []);
}
```

```
return (  
  <>  
  <h1>Current BTC/USD data</h1>  
  <p>Code: {btcData.code}</p>  
  <p>Symbol: {btcData.symbol}</p>  
  <p>Rate: {btcData.rate}</p>  
  <p>Description: {btcData.description}</p>  
  <p>Rate Float: {btcData.rate_float}</p>  
  </>  
) ;  
}
```

This example shows that in order to fetch data from a third party API, you need to pass an anonymous function as a call to the `useEffect` hook.

```
1  
2  
3  
4  
5  
6  
  
useEffect(  
  () => {  
    // ... data fetching code goes here
```

```
},  
[]  
) ;
```

The code above emphasizes the fact that the `useEffect` hook takes two arguments, and that the first argument holds the anonymous function, which, inside its body, holds the data fetching code. Alternatively, you might extract this anonymous function into a separate function expression or function declaration, and then just reference it.

Using the above example, that code could be presented as follows:

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

```
17
18
19
20
21
22
23
24
25
26
27

import { useState, useEffect } from "react";

export default function App() {
  const [btcData, setBtcData] = useState({});

  const fetchData = () => {
    fetch(`https://api.coindesk.com/v1/bpi/currentprice.json`)
      .then((response) => response.json())
      .then((jsonData) => setBtcData(jsonData.bpi.USD))
      .catch((error) => console.log(error));
  }
}
```

```

} ;

useEffect(() => {

  fetchData();

}, []);

return (
<>

<h1>Current BTC/USD data</h1>

<p>Code: {btcData.code}</p>

<p>Symbol: {btcData.symbol}</p>

<p>Rate: {btcData.rate}</p>

<p>Description: {btcData.description}</p>

<p>Rate Float: {btcData.rate_float}</p>

</>

);
}

```

The code essentially does the same thing, but this second example is cleaner and better organized. One additional thing that can be discussed here is the `return` statement of the above example. Very often, the response from fetching third-party data might fail, or be delayed. That's why it can be useful to provide different renders, based on whether or not the data has been received. The simplest conditional rendering might involve setting up two renders, based on whether or not the data has been successfully fetched. For example:

```

2
3
4
5
6
7
8

return someStateVariable.length > 0 ? (
  <div>
    <h1>Data returned:</h1>
    <h2>{someStateVariable.results[0].price}</h2>
  </div>
) : (
  <h1>Data pending...</h1>
);

```

In this example, I'm conditionally returning an `h1` and `h2`, if the length of the `someStateVariable` binding's length is longer than 0.

This approach would work if the `someStateVariable` holds an array.

If the `someStateVariable` is initialized as an empty array, passed to the call to the `useState` hook, then it would be possible to update this state variable with an array item that might get returned from a `fetch()` call to a third-party JSON data provider.

If this works out as described above, the length of the `someStateVariable` would increase from the starting length of zero - because an empty array's length is zero.

Let's inspect the conditional `return` again:

1

2

3

4

5

6

7

8

```
return someStateVariable.length > 0 ? (
  <div>
    <h1>Data returned:</h1>
    <h2>{someStateVariable.results[0].price}</h2>
  </div>
) : (
  <h1>Data pending...</h1>
);
```

If the data fetching fails, the text of "Data pending..." will render on the screen, since the length of the `someStateVariable` will remain being zero.

Conclusion

You learned more about fetching data using hooks and that fetching data from a third-party API is considered a side-effect that requires the use of the `useEffect` hook to deal with the Fetch API calls in React.

You also explored how the response from fetching third-party data might fail, or be delayed, and that it can be useful to provide different renders, based on whether or not the data has been received.

Completed

Solution: Can you fetch data?

Here is the completed solution code for the `App.js` file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

```
import React from "react";  
  
function App() {  
  
  const [user, setUser] = React.useState([]);  
  
  const fetchData = () => {  
  
    fetch("https://randomuser.me/api/?results=1")  
      .then((response) => response.json())  
      .then((data) => setUser(data));  
  
  };  
  
  React.useEffect(() => {  
    fetchData();  
  }, []);  
}  
  
export default App;
```

```

}, []);  
  

return Object.keys(user).length > 0 ? (  


# Customer data


## Name: {user.results[0].name.first}

</div>  

)> : (  

<h1>Data pending...</h1>  

);  

}  
  

export default App;

```

Here is the sample output from the solution code for the `App.js` file. Note that the lab produces a different random image and name with each refresh, so your output is likely to have a different name and a different customer image than the one shown below.

Customer data

Name: Liam



Step-by-step solution

Step 1

Inside the `fetchData()` function's code block, you executed the `fetch()` function, passing it a single string argument: "`https://randomuser.me/api/?results=1`".

1

2

3

```
const fetchData = () => {  
  
  fetch("https://randomuser.me/api/?results=1")  
  
};
```

Step 2

Next, inside the `fetchData()` function, under the `fetch()` function call, you added the following piece of code:

1

2

3

4

```
const fetchData = () => {  
  
  fetch("https://randomuser.me/api/?results=1")  
  
    .then((response) => response.json())  
  
};
```

Step 3

You then added another `then()` call, which takes an arrow function.

The passed-in arrow function, receives a `data` argument, and using that `data` argument, it invokes the `setUser()` function, with the data passed to it.

1

2

3

4

5

```
const fetchData = () => {  
  
  fetch("https://randomuser.me/api/?results=1")  
  
  .then((response) => response.json())  
  
  .then((data) => setUser(data));  
  
};
```

Step 4

In the `return` statement of the App component, under the `h1` heading that reads “Customer data”, you added an `h2` heading, with the following code: `Name: {user.results[0].name.first}`

1

2

3

4

5

6

7

8

```
return Object.keys(user).length > 0 ? (  
  
  <div style={{padding: "40px"}}>  
  
    <h1>Customer data</h1>
```

```
<h2>Name: {user.results[0].name.first}</h2>
</div>

) : (

<h1>Data pending...</h1>

);
```

Step 5

You then updated the `return` statement of the App component by adding another line of code under the newly-added `h2`.

You added an `img` element, with the `src` attribute and an `alt` attribute holding the following code:
`{user.results[0].picture.large}`

```
1
2
3
4
5
6
7
8
9
```

```
return Object.keys(user).length > 0 ? (
  <div style={{padding: "40px"}}>
    <h1>Customer data</h1>
    <h2>Name: {user.results[0].name.first}</h2>
```

```
<img src={user.results[0].picture.large} alt="" />
</div>

) : (

<h1>Data pending...</h1>

);
```

Completed

Additional resources

Below is a list of additional resources as you continue to explore React hooks and custom hooks. In particular, to complement your learning on the 'Rules of hooks and fetching data with hooks' lesson, you can work through the following:

- The [Rules of Hooks reading on Reactjs.org](#) website gives you an overview of how to work with the hooks as recommended by the React Core team at Meta.
- The [Fetching data with Effects](#) article on React docs discusses fetching data using a few different approaches, including using `async / await` syntax.
- [How to use promises](#) is a resource that describes the "behind-the-scenes" of how data fetching works in greater depth.
- [async function](#) is a resource on MDN that discusses the use of the `async` and `await` keywords as a more recent way to handle API requests in JavaScript.

Completed

When to choose `useReducer` vs `useState`

The `useState` hook is best used on less complex data.

While it's possible to use any kind of a data structure when working with `useState`, it's better to use it with primitive data types, such as strings, numbers, or booleans.

The `useReducer` hook is best used on more complex data, specifically, arrays or objects.

While this rule is simple enough, there are situations where you might be working with a simple object and still decide to use the `useState` hook.

Such a decision might stem from the simple fact that working with `useState` can sometimes feel easier than thinking about how the state is controlled when working with `useReducer`.

It might help conceptualizing this dilemma as a gradual scale, on the left side of which, there is the `useState` hook with primitive data types and simple use cases, such as toggling a variable on or off. At the end of this spectrum, there is the `useReducer` hook used to control state of large state-holding objects.

There's no clear-cut point on this spectrum, at which point you would decide: "If my state object has three or more properties, I'll use the `useReducer` hook".

Sometimes such a statement might make sense, and other times it might not.

What's important to remember is to keep your code simple to understand, collaborate on, contribute to, and build from.

One negative characteristic of `useState` is that it often gets hard to maintain as the state gets more complex.

On the flip side, a negative characteristic of `useReducer` is that it requires more prep work to begin with. There's more setup involved. However, once this setup is completed, it gets easier to extend the code based on new requirements.

Conclusion

You learned about the decision-making process when choosing between `useReducer` and `useState` for working with different types of data.

Completed

Custom hooks

React has some built-in hooks, such as the `useState` hook, or the `useRef` hook, which you learned about earlier. However, as a React developer, you can write your own hooks. So, why would you want to write a custom hook?

In essence, hooks give you a repeatable, streamlined way to deal with specific requirements in your React apps. For example, the `useState` hook gives us a reliable way to deal with state updates in React components.

A custom hook is simply a way to extract a piece of functionality that you can use again and again. Put differently, you can code a custom hook when you want to avoid duplication or when you do not want to build a piece of functionality from scratch across multiple React projects. By coding a custom hook, you can create a reliable and streamlined way to reuse a piece of functionality in your React apps.

To understand how this works, let's explore how to build a custom hook. To put this in context, let's also code a very simple React app.

The entire React app is inside the App component below:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
import { useState } from "react";
```

```

function App() {

  const [count, setCount] = useState(0);

  function increment() {
    setCount(prevCount => prevCount + 1)
  }

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Plus 1</button>
    </div>
  );
}

export default App;

```

This is a simple app with an `h1` heading that shows the value of the `count` state variable and a button with an `onClick` event-handling attribute which, when triggered, invokes the `increment()` function. The hook will be simple too. It will console log a variable's value whenever it gets updated. Remember that the proper way to handle `console.log()` invocations is to use the `useEffect` hook.

So, this means that my custom hook will:

1. Need to use the `useEffect` hook and
2. Be a separate file that you'll then use in the `App` component.

How to name a custom hook

A custom hook needs to have a name that begins with `use`.
Because the hook in this example will be used to log values to the console, let's name the hook `useConsoleLog`.

Coding a custom hook

Now's the time to explore how to code the custom hook.

First, you'll add it as a separate file, which you can name `useConsoleLog.js`, and add it to the root of the `src` folder, in the same place where the `App.js` component is located.

Here's the code of the `useConsoleLog.js` file:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
  
import { useEffect } from "react";  
  
function useConsoleLog(varName) {  
  useEffect(() => {  
    console.log(varName);  
  }, [varName]);  
}
```

```
export default useConsoleLog;
```

Using a custom hook

Now that the custom hook has been coded, you can use it in any component in your app. Since the app in the example only has a single component, named App, you can use it to update this component.

The `useConsoleLog` hook can be imported as follows:

```
import useConsoleLog from "./useConsoleLog";
```

And then, to use it, under the state-setting code, I'll just add the following line of code:

```
useConsoleLog(count);
```

Here's the completed code of the App.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

```
import { useState } from "react";
import useConsoleLog from "./useConsoleLog";

function App() {
  const [count, setCount] = useState(0);
  useConsoleLog(count);

  function increment() {
    setCount(prevCount => prevCount + 1);
  }

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Plus 1</button>
    </div>
  );
}
```

```
}
```

```
export default App;
```

This update confirms the statement made at the beginning of this reading, which is that custom hooks are a way to extract functionality that can then be reused throughout your React apps

Conclusion

You have learned how to name, build and use custom hooks in React.

Completed

Solution: Create your own custom hook, usePrevious

Your task was to complete the custom hook named `usePrevious` so that the `h1` heading shows both the current day and the previous current day before the update.

Here is the completed solution code for the `App.js` file:

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import { useState, useEffect, useRef } from "react";

export default function App() {

  const [day, setDay] = useState("Monday");

  const prevDay = usePrevious(day);

  const getNextDay = () => {

    if (day === "Monday") {

      setDay("Tuesday")

    } else if (day === "Tuesday") {

      setDay("Wednesday")

    } else if (day === "Wednesday") {

      setDay("Thursday")

    } else if (day === "Thursday") {
```

```
        setDay("Friday")

    } else if (day === "Friday") {

        setDay("Monday")

    }

}

return (
<div style={{padding: "40px"}}>

<h1>

    Today is: {day}<br />

{

    prevDay && (

        <span>Previous work day was: {prevDay}</span>

    )

}

</h1>

<button onClick={getNextDay}>

    Get next day

</button>

</div>

) ;

}
```

```
function usePrevious(val) {  
  
  const ref = useRef();  
  
  useEffect(() => {  
  
    ref.current = val;  
  
  }, [val]);  
  
  return ref.current;  
  
}
```

Steps

Step 1

You should have located the `usePrevious()` function.

1

2

3

```
function usePrevious(val) {  
  
}  
  
}  
  
}

```
function usePrevious(val) {

 const ref = useRef();

 useEffect(() => {

 ref.current = val;

 }, [val]);

 return ref.current;

}
```


```

Inside the `usePrevious()` function's code block, you needed to invoke the `useRef` hook without any arguments, and assign this invocation to a variable named `ref`, declared using the `const` keyword.

1

2

3

```
function usePrevious(val) {  
  
  const ref = useRef();  
  
  useEffect(() => {  
  
    ref.current = val;  
  
  }, [val]);  
  
  return ref.current;  
  
}
```

```
}
```

Step 2

Next, inside the `usePrevious()` function declaration, you needed to add a call to the `useEffect()` hook.

1

2

3

4

```
function usePrevious(val) {  
  
  const ref = useRef();  
  
  useEffect();  
  
}
```

Step 3

Then, you needed to pass two parameters as arguments to the `useEffect()` function call. The first parameter should have been an arrow function, without any arguments. Inside the arrow function's body, you should have assigned the `val` value to the current property on the `ref` object. The second parameter needed to be the dependencies array. The dependencies array needed to list a single variable - namely, the `val` variable.

1

2

3

4

5

6

```
function usePrevious(val) {
```

```
const ref = useRef();

useEffect(() => {
  ref.current = val;
}, [val]);
```

Step 4

You needed to add one more line to the body of the `usePrevious()` function declaration, to specify the `return` value of that function.

The `usePrevious()` function should have returned the `ref.current` value, as follows:

```
1
2
3
4
5
6
7

function usePrevious(val) {
  const ref = useRef();
  useEffect(() => {
    ref.current = val;
  }, [val]);
  return ref.current;
}
```

Completed

Additional resources

Below is a list of additional readings as you continue to explore 'React hooks and custom hooks'. In particular, to complement your learning in the 'Advanced hooks' lesson, you can work through the following:

- The [useReducer hook reference](#) in the React docs discusses the basics of `useReducer`, along with specifying initial state and lazy initialization.
- The React docs also has a reference on [using the useRef hook](#) which is a great example of various options that are available when working with the `useRef` hook.
- The [Reusing Logic with Custom Hooks](#) reference in the React docs discusses the dynamics of custom hooks and provides a few practical examples to complement the theory behind them.

Completed

Week3-

Types of Children

In JSX expressions, the content between an opening and closing tag is passed as a unique prop called children. There are several ways to pass children, such as rendering string literals or using JSX elements and JavaScript expressions. It is also essential to understand the types of JavaScript values that are ignored as children and don't render anything. Let's explore these in a bit more detail:

String literals

String literals refer to simple JavaScript strings. They can be put between the opening and closing tags, and the `children` prop will be that string.

```
<MyComponent>Little Lemon</MyComponent>
```

In the above example, the `children` prop in `MyComponent` will be simply the string "Little Lemon". There are also some rules JSX follows regarding whitespaces and blank lines you need to bear in mind, so that you understand what to expect on your screen when those edge cases occur.

1. JSX removes whitespaces at the beginning and end of a line, as well as blank lines:

1

2

3

4

```
<div>      Little Lemon      </div>

<div>
  Little Lemon
</div>
```

2. New lines adjacent to tags are removed:

1

2

3

4

```
<div>

  Little Lemon
</div>
```

3. JSX condenses new lines that happen in the middle of string literals into a single space:

1

2

3

```
<div>
  Little
  Lemon
</div>
```

That means that all the instances above render the same thing.

JSX Elements

You can provide JSX elements as children to display nested components:

```
<Alert>
  <Title />
  <Body />
</Alert>
```

JSX also enables mixing and matching different types of children, like a combination of string literals and JSX elements:

```
<Alert>
```

```
<div>Are you sure?</div>

<Body />

</Alert>
```

A React component can also return a bunch of elements without wrapping them in an extra tag. For that, you can use React Fragments either using the explicit component imported from React or empty tags, which is a shorter syntax for a fragment. A React Fragment component lets you group a list of children without adding extra nodes to the DOM. You can learn more about fragments in the additional resources unit from this lesson.

The two code examples below are equivalent, and it's up to your personal preference what to choose, depending on whether you prefer explicitness or a shorter syntax:

```
1
2
3
4
5
6
7
8
9
10
11
12
13

return (
  <React.Fragment>
```

```
<li>Pizza margarita</li>

<li>Pizza diavola</li>

</React.Fragment>

);

return (
<>
<li>Pizza margarita</li>

<li>Pizza diavola</li>

</>
);


```

JavaScript Expressions

You can pass any JavaScript expression as children by enclosing it within curly braces, {}. The below expressions are identical:

```
<MyComponent>Little Lemon</MyComponent>
<MyComponent>{'Little Lemon'}</MyComponent>
```

This example is just for illustration purposes. When dealing with string literals as children, the first expression is preferred.

Earlier in the course, you learned about lists. JavaScript expressions can be helpful when rendering a list of JSX elements of arbitrary length:

1

2

3

4

5

```

6
7
8
9
10
11
12

function Dessert(props) {
  return <li>{props.title}</li>;
}

function List() {
  const desserts = ['tiramisu', 'ice cream', 'cake'];
  return (
    <ul>
      {desserts.map((dessert) => <Item key={dessert} title={dessert} />)}
    </ul>
  );
}

```

Also, you can mix JavaScript expressions with other types of children without having to resort to string templates, like in the example below:

1

2

3

```
function Hello(props) {  
  
  return <div>Hello {props.name} !</div>;  
  
}
```

Functions

Suppose you insert a JavaScript expression inside JSX. In that case, React will evaluate it to a string, a React element, or a combination of the two. However, the children prop works just like any other prop, meaning it can be used to pass any type of data, like functions.

Function as children is a React pattern used to abstract shared functionality that you will see in detail in the next lesson.

Booleans, Null and Undefined, are ignored

false, null, undefined, and true are all valid children. They simply don't render anything. The below expressions will all render the same thing:

```
<div />  
<div></div>  
<div>{false}</div>  
<div>{null}</div>  
<div>{undefined}</div>  
<div>{true}</div>
```

Again, this is all for demonstration purposes so that you know what to expect on your screen when these special values are used in your JSX.

When used in isolation, they don't offer any value. However, boolean values like true and false can be useful to conditionally render React elements, like rendering a Modal component only if the variable `showModal` is true

1

2

3

```
<div>  
  
  {showModal && <Modal />}  
  
</div>
```

However, keep in mind that React still renders some "false" values, like the 0 number. For example, the below code will not behave as you may expect because 0 will be printed when `props.desserts` is an empty array:

```
1
2
3
4
5

<div>
  {props.desserts.length &&
    <DessertList desserts={props.desserts} />
  }
</div>
```

To fix this, you need to make sure the expression before `&&` is always boolean:

```
1
2
3
4
5
6
7
8
9
```

[10](#)

[11](#)

```
<div>

{props.desserts.length > 0 &&

<DessertList desserts={props.desserts} />

}

</div>
```

```
<div>

{!!props.desserts.length &&

<DessertList desserts={props.desserts} />

}

</div>
```

Conclusion

You have learned about different types of children in JSX, such as how to render string literals as children, how JSX elements and JavaScript expressions can be used as children, and the boolean, null or undefined values that are ignored as children and don't render anything.

Completed

Solution: Build a Radio Group Component

Here is the completed solution code for the `Radio/index.js` file:

[1](#)

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

```
import * as React from "react";

export const RadioGroup = ({ onChange, selected, children }) => {

  const RadioOptions = React.Children.map(children, (child) => {

    return React.cloneElement(child, {
      onChange,
      checked: child.props.value === selected,
    });
  });

  return <div className="RadioGroup">{RadioOptions}</div>;
};

export const RadioOption = ({ value, checked, onChange, children }) => {

  return (
    <div>
      <input checked={checked} type="radio" value={value} />
      {children}
    </div>
  );
}
```

```

<div className="RadioOption">

  <input
    id={value}
    type="radio"
    name={value}
    value={value}
    checked={checked}
    onChange={(e) => {
      onChange(e.target.value);
    }}
  />

  <label htmlFor={value}>{children}</label>
</div>

);
}

```

Step 1

The API for the `RadioGroup` component is defined as two props: `selected`, which is a string that matches one of the `RadioOption` values and `onChange`, which is the event that gets called whenever a selection changes, providing the new value as an argument.

1

2

3

4

5

6

```
<RadioGroup onChange={setSelected} selected={selected}>

  <RadioOption value="social_media">Social Media</RadioOption>

  <RadioOption value="friends">Friends</RadioOption>

  <RadioOption value="advertising">Advertising</RadioOption>

  <RadioOption value="other">Other</RadioOption>

</RadioGroup>
```

Step 2

The `RadioOptions` variable should be assigned to the return value of `React.Children.map`, which will be a new React element. The first argument passed to the map function should be the `children` prop, and the second is a function that gets invoked in every `child` contained within the `children` property. Recall that a `children` prop is a special prop all React components have and that it presents a special data structure, similar to arrays, where you can perform iterations. However, they are not exactly instances of JavaScript arrays. That's why to iterate over all siblings you should use the special `React.children.map` API provided by React.

Inside the map projection function, you should first clone the element using `React.cloneElement`, passing as first argument the target `child` element and as a second argument a configuration with all new props. The resulting element will have the original element's props with the new props merged in.

`onChange` can be passed to each child (`RadioOption`) as it is and checked is the property the `RadioOption` uses to determine if the underlying `radio` input is selected. Since `RadioGroup` receives a `selected` property, which is a string pointing to the value of the option that has been selected, `checked` will be only true for one of the options at any point in time. This is guaranteed by performing an equality check, comparing the `RadioOption` value prop with the selected value. Finally, the `RadioGroup` component returns the new `RadioOptions` elements by wrapping them in curly braces.

1

2

3

4

```
5
6
7
8
9
10
11

import * as React from "react";

export const RadioGroup = ({ onChange, selected, children }) => {
  const RadioOptions = React.Children.map(children, (child) => {
    return React.cloneElement(child, {
      onChange,
      checked: child.props.value === selected,
    });
  });
  return <div className="RadioGroup">{RadioOptions}</div>;
};


```

Step 3

The `RadioOption` component now receives two new props implicitly, `onChange` and `checked`, that `RadioGroup` is injecting via children manipulation, as seen in the previous section. The `value` prop is already provided explicitly inside the `App.js` component and `children` represents the label text for the radio input.

You have to connect the props `value`, `checked` and `onChange` correctly. First, both `value` and `checked` props should be passed to the `radio` input as is. Then, you should use the `onChange` event from the `radio` input, retrieve the `value` property from the event target object and pass it to the `onChange` prop as the argument as seen below. That completes the implementation of the `RadioOption` component.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

export const RadioOption = ({ value, checked, onChange, children }) => {
  return (
    <div>
      <input checked={checked} type="radio" value={value} />
      {children}
    </div>
  )
}
```

```

<div className="RadioOption">

  <input
    id={value}
    type="radio"
    name={value}
    value={value}
    checked={checked}
    onChange={(e) => {
      onChange(e.target.value);
    }}
  />

  <label htmlFor={value}>{children}</label>
</div>

);

```

Step 4

Once you run the application in the browser, you should see something similar to the screenshot below.

The important thing is that the button should be enabled as soon as a selection is made. Don't worry if nothing happens when you click it, it's intended. In this exercise, the button click event has no action bound to it.

How did you hear about Little Lemon?

- Social Media
- Friends
- Advertising
- Other

Submit

Completed

Additional resources

Here is a list of additional resources for JSX Deep Dive:

- [Chakra-UI](#) is an open-source component library that embraces all the concepts explained during this lesson, being a nice option if you would like to start your project with a set of atomic components that have been carefully designed with flexibility in mind, so that they can be customized as per your theme requirements.
- [Compound components with hooks](#) is an article that illustrates how a combination of component composition, context and hooks can lead to a clean and concise component design.
- [React Fragments](#) from the official React docs illustrates how to group a list of React children without adding extra nodes to the DOM.

Completed

Higher-order components

In a previous lesson, you learned about Higher-order components (HOC) as a pattern to abstract shared behavior, as well as a basic example of an implementation.

Let's dive deeper to illustrate some of the best practices and caveats regarding HOCs.

These include never mutating a component inside a HOC, passing unrelated props to your wrapped component, and maximizing composability by leveraging the `Component => Component` signature.

Don't mutate the original component

One of the possible temptations is to modify the component that is provided as an argument, or in other words, mutate it. That's because JavaScript allows you to perform such operations, and in some cases, it seems the most straightforward and quickest path. Remember that React promotes immutability in all scenarios. So instead, use composition and turn the HOC into a pure function that does not alter the argument it receives, always returning a new component.

```
1  
2  
3  
4  
5  
6  
7  
  
const HOC = (WrappedComponent) => {  
  // Don't do this and mutate the original component  
  WrappedComponent = () => {  
    ...  
  };  
}  
}
```

Pass unrelated props through to the Wrapped Component

HOC adds features to a component. In other words, it enhances it. That's why they shouldn't drastically alter their original contract. Instead, the component returned from a HOC is expected to have a similar interface to the wrapped component.

HOCs should spread and pass through all the props that are unrelated to their specific concern, helping ensure that HOCs are as flexible and reusable as possible, as demonstrated in the example below:

1

2

3

4

5

6

7

```
const with.mousePosition = (WrappedComponent) => {  
  const injectedProp = {mousePosition: {x: 10, y: 10}};  
  
  return (originalProps) => {  
    return <WrappedComponent injectedProp={injectedProp}  
    {...originalProps} />;  
  };  
};
```

Maximize composability

So far, you have learned that the primary signature of a HOC is a function that accepts a React component and returns a new component.

Sometimes, HOCs can accept additional arguments that act as extra configuration determining the type of enhancement the component receives.

1

```
const EnhancedComponent = HOC(WrappedComponent, config)
```

The most common signature for HOCs uses a functional programming pattern called "currying" to maximize function composition. This signature is used extensively in React libraries, such as [React Redux](#), which is a popular library for managing state in React applications.

1

```
const EnhancedComponent = connect(selector, actions)(WrappedComponent);
```

This syntax may seem strange initially, but if you break down what's happening separately, it would be easier to understand.

1

2

```
const HOC = connect(selector, actions);

const EnhancedComponent = HOC(WrappedComponent);
```

`connect` is a function that returns a higher-order component, presenting a valuable property for composing several HOCs together.

Single-argument HOCs like the ones you have explored so far, or the one returned by the `connect` function has the signature `Component => Component`. It turns out that functions whose output type is the same as its input type are really easy to compose together.

1

2

3

4

5

6

7

8

9

```
const enhance = compose (

  // These are both single-argument HOCs

  with.mousePosition,
  with.URLLocation,
  connect(selector)
```

```
) ;
```

```
// Enhance is a HOC  
  
const EnhancedComponent = enhance(WrappedComponent);
```

Many third-party libraries already provide an implementation of the compose utility function, like [lodash](#), [Redux](#), and [Ramda](#). Its signature is as follows:

`compose(f, g, h)` is the same as `(...args) => f(g(h(...args)))`

Caveats

Higher-order components come with a few caveats that aren't immediately obvious.

1. Don't use HOCs inside other components: always create your enhanced components outside any component scope. Otherwise, if you do so inside the body of other components and a re-render occurs, the enhanced component will be different. That forces React to remount it instead of just updating it. As a result, the component and its children would lose their previous state.

1

2

3

4

5

6

7

8

9

10

11

```
const Component = (props) => {
```

```

// This is wrong. Never do this

const EnhancedComponent = HOC(WrappedComponent);

return <EnhancedComponent />

};

// This is the correct way

const EnhancedComponent = HOC(WrappedComponent);

const Component = (props) => {

  return <EnhancedComponent />

};

```

1. Refs aren't passed through: since React `refs` are not `props`, they are handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component. To solve this, you can use the [React.forwardRef API](#). You can learn more about this API and its use cases in the additional resources section from this lesson.

Conclusion

And in summary, you have examined higher-order components in more detail. The main takeaways are never mutating a component inside a HOC and passing unrelated props to your wrapped component.

You also learned how to maximize composability by leveraging the `Component => Component` signature and addressed some caveats about HOC.

Completed

Solution: Implementing scroller position with render props

Here is the completed solution code for the App.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

```
import "./App.css";
```

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import { useEffect, useState } from "react";

const MousePosition = ({ render }) => {

  const [mousePosition, setMousePosition] = useState({
    x: 0,
    y: 0,
  });

  useEffect(() => {
    const handleMousePositionChange = (e) => {
      setMousePosition({
        x: e.clientX,
        y: e.clientY,
      });
    };

    window.addEventListener("mousemove", handleMousePositionChange);

    return () => {
      window.removeEventListener("mousemove", handleMousePositionChange);
    };
  });
}
```

```

}, []);

return render({ mousePosition });
};

const PanelMouseLogger = () => {
  return (
    <div className="BasicTracker">
      <p>Mouse position:</p>
      <.mousePosition
        render={({ mousePosition }) => (
          <div className="Row">
            <span>x: {mousePosition.x}</span>
            <span>y: {mousePosition.y}</span>
          </div>
        ) }
      />
    </div>
  );
}

```

1. Implement the body of handleMousePositionChange

The `mousemove` handler function receives an event as parameter that contains the mouse coordinates as `clientX` and `clientY` properties. Therefore you can provide a position update by calling the state setter `setMousePosition` with the new values.

2

3

4

5

6

7

```
const handleMousePositionChange = (e) => {  
  
  setMousePosition({  
  
    x: e.clientX,  
  
    y: e.clientY,  
  
  }) ;  
  
};
```

2. Implement the `return` statement of the component

The `MousePosition` component receives a `render` prop, which is the special prop name designed by convention to specify a function that returns some JSX. Since the `MousePosition` component does not take care of any visualization logic, but rather encapsulates cross-cutting concerns, it should return the result of calling the `render` function with the `mousePosition` as an argument. In other words, it's up to the components that consume `MousePosition` to specify what sort of UI they want to display when they receive a new value of the mouse position on the screen.

1

```
return render({ mousePosition })
```

Step 2

The `PanelMouseLogger` component should not receive any props. Hence, the early return from the previous implementation if no props were provided is no longer needed.

Instead, the `mousePosition` is now injected as the first argument of the render function prop that `MousePosition` uses. It's in this render function body where the previous JSX should be extracted and returned.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

const PanelMouseLogger = () => {

  return (
    <div className="BasicTracker">
      <p>Mouse position:</p>
    </div>
  )
}
```

```

<.mousePosition

  render={({ mousePosition }) => (
    <div className="Row">
      <span>x: {mousePosition.x}</span>
      <span>y: {mousePosition.y}</span>
    </div>
  )}

/>

</div>

);

}

```

Step 3

Similarly, as in step 2, the component should not receive any props and the early if statement should be removed. The particular UI for this component is provided as part of the render prop as well.

[1](#)

[2](#)

[3](#)

[4](#)

[5](#)

[6](#)

[7](#)

[8](#)

9

10

11

12

```
const PointMouseLogger = () => {

  return (
    <.mousePosition
      render={({ mousePosition }) => (
        <p>
          ({mousePosition.x}, {mousePosition.y})
        </p>
      ) }
    />
  );
};
```

At this point, the implementation has been completed and you should see the following result when you run the app in the browser:

Little Lemon Restaurant

Mouse position:

x: 217 y: 432

(217, 432)

Completed

Additional resources

Here is a list of additional resources as you continue to explore Reusing Behavior:

- [Downshift](#) is a popular open-source library that implements an autocomplete, combo box or select experience using the `render` prop pattern.
- [Render props](#) from the official React docs.
- [Higher Order Components](#) from the official React docs.

- [Forwarding Refs](#) from the official React docs showcases in detail how to forward refs in higher-order components, so that they are passed through properly.

Completed

Solution: Writing more test scenarios

Here is the completed solution code for the App.test.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import { fireEvent, render, screen } from "@testing-library/react";
import FeedbackForm from "./FeedbackForm";

describe("Feedback Form", () => {
  test("User is able to submit the form if the score is lower than 5 and additional feedback is provided", () => {
    const score = "3";
    const comment = "The pizza crust was too thick";
    const handleSubmit = jest.fn();
    render(<FeedbackForm onSubmit={handleSubmit} />);

    const rangeInput = screen.getByLabelText(/Score:/);
    fireEvent.change(rangeInput, { target: { value: score } });

    const textArea = screen.getByLabelText(/Comments:/);
    fireEvent.change(textArea, { target: { value: comment } });
  });
});
```

```
const submitButton = screen.getByRole("button");

fireEvent.click(submitButton);

expect(handleSubmit).toHaveBeenCalledWith({
  score,
  comment,
}) ;

}) ;

test("User is able to submit the form if the score is higher than 5, without additional feedback", () => {

  const score = "9";

  const handleSubmit = jest.fn();

  render(<FeedbackForm onSubmit={handleSubmit} />);

  const rangeInput = screen.getByLabelText(/Score:/);

  fireEvent.change(rangeInput, { target: { value: score } });

  const submitButton = screen.getByRole("button");

  fireEvent.click(submitButton);
```

```

expect(handleSubmit).toHaveBeenCalledWith(
  score,
  comment: ""
);

```

Steps

Step 1

The first test scenario has the following specification:

User is able to submit the form if the score is lower than 5 and additional feedback is provided

The test scenario already contains some initial code that acts as boilerplate before getting to the bulk of the test, in particular:

- Two variables that hold the desired state of the form, a score of 3 and an additional comment.
- A mock function that is called when submitting the form.
- The rendering of the form component.
- The final assertion that should make the test pass.

If you run as it is, the test will fail stating that the mock function has not been called at all. That is because no interactions have occurred yet and it's your task to write those.

```

expect(jest.fn()).toHaveBeenCalled(...expected)

Expected: {"comment": "The pizza crust was too thick", "score": "3"}

Number of calls: 0

11 |     // You have to write the rest of the test below to make the assertion pass
12 |
> 13 |     expect(handleSubmit).toHaveBeenCalled(
    |     ^
14 |     score,
15 |     comment,
16 | );

```

at Object.<anonymous> (src/App.test.js:13:26)

The first user interaction that needs to happen is to set the score as 3. The following code achieves that:

1

2

```
const rangeInput = screen.getByLabelText(/Score:/);
```

```
fireEvent.change(rangeInput, { target: { value: score } });
```

The first line grabs a reference to the range input component by using the global screen object from react-testing-library and the query `getByLabelText` to find a label that contains the exact text `Score`:

Then, a change event is simulated on the input, passing as the event an object with the `value` property set to the variable `score: event.target.value = score`

After that, a second user interaction is required to set the additional comment. This is the code that accomplishes that:

1

2

```
const textArea = screen.getByLabelText(/Comments:/);  
  
fireEvent.change(textArea, { target: { value: comment } });
```

No further explanation is needed regarding those two lines, since they mimic the same interaction as with the range input.

Last but not least, a submission of the form should be simulated by calling the below two lines:

1

2

```
const submitButton = screen.getByRole("button");  
  
fireEvent.click(submitButton);
```

In this particular instance, the button is referenced by using a different query on the global screen object, `getByRole`. This query looks for an element whose role attribute is set to "button", which is inherent in all button HTML tags.

The form is finally submitted via firing a click event on the button instance.

If you run the command `npm test` in your terminal, the test should pass now.

Let's now cover the second scenario.

User is able to submit the form if the score is higher than 5, without additional feedback

The below represents the code you need to write to make the test pass.

1

2

3

4

```

const rangeInput = screen.getByLabelText(/Score:/);

fireEvent.change(rangeInput, { target: { value: score } });

const submitButton = screen.getByRole("button");

fireEvent.click(submitButton);

```

This test is simpler and there is nothing new besides skipping any interaction with the text area, since no additional feedback is required when the score provided is higher than 5, being 9 in this scenario.

Step 2

If you run `npm test` after adding the lines above, both of your tests should pass successfully, with the terminal providing the below output.

```

PASS | src/App.test.js
Feedback Form
  ✓ User is able to submit the form if the score is lower than 5 and additional feedback is provided (78 ms)
  ✓ User is able to submit the form if the score is higher than 5, without additional feedback (20 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.379 s
Ran all test suites related to changed files.

Watch Usage
> Press a to run all tests.
> Press f to run only failed tests.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.

```

Completed

Introduction to continuous integration

Introduction

Continuous Integration (CI) is a software development technique in which developers use a version control system, like Git, and push code changes daily, multiple times a day. Instead of building out features in isolation and integrating them at the end of the development cycle, a more iterative approach is employed.

Each merge triggers an automated set of scripts to automatically build and test your application. These scripts help decrease the chances that you introduce errors in your application.

If some of the scripts fail, the CI system doesn't progress to further stages, issuing a report that developers can use to promptly assess what was wrong and resolve the problem.

This reading will teach you why embracing a CI tool is essential for your software development process. You will also explore a typical development workflow that you can integrate into your CI system and some of the main benefits of using CI.

Why do we need CI?

In new product development, the time to figure everything out up front is limited. Taking smaller steps helps estimate more accurately and validate more often. Having a shorter feedback loop involves more iterations. This number of iterations, not the number of hours invested, drives the process forward.

Working in long feedback loops is risky for software development teams, increasing the chances of introducing errors. Integrating new changes is also time-consuming.

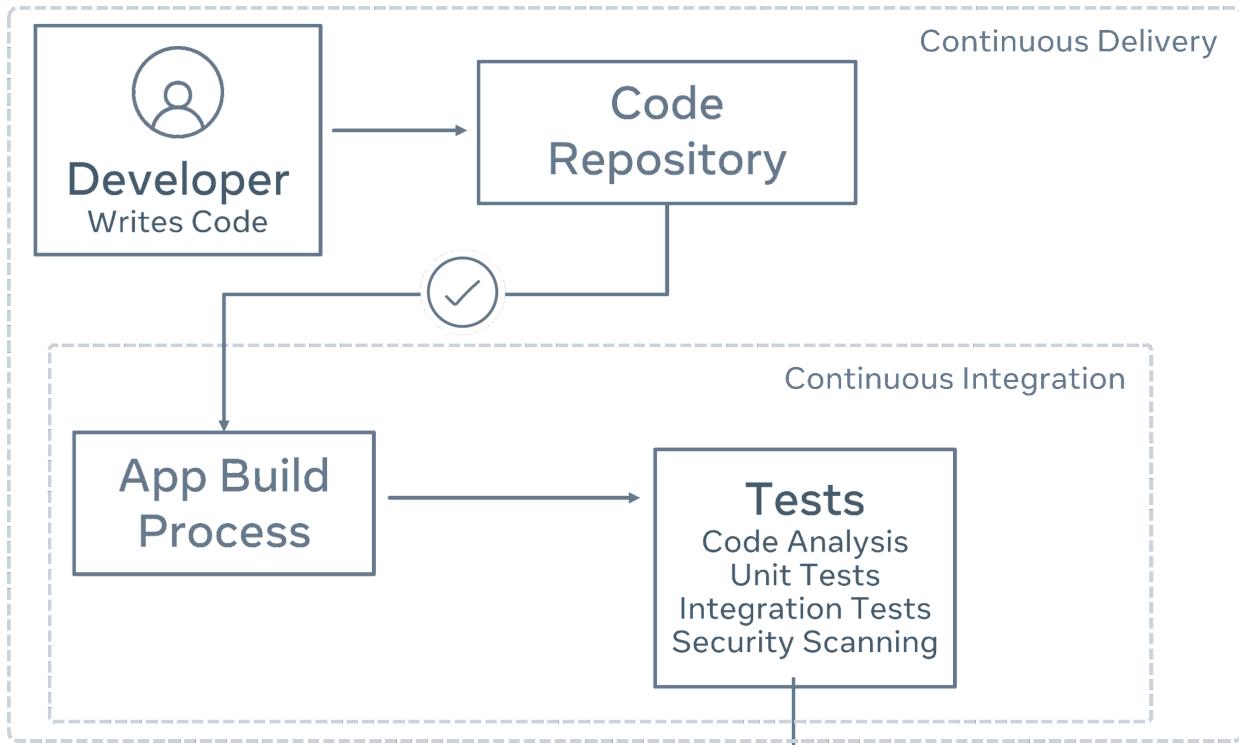
By automating all integration steps and having small controlled changes, developers avoid repetitive work and minimize human errors. The CI tool monitors the central code repository and prevents people from deciding when and how to run tests. Every time there is a new commit, it runs all automated tests.

Based on the outcome, it either accepts the commit if all tests passed successfully or reject it if there was a failure.

CI Pipeline

Below is a graphical representation of a typical CI process as a pipeline. When new code enters one end, a new version of the app gets built automatically, and a suite of automated tests is run against it.

Continuous Integration is a small part of a more significant process called Continuous Delivery. However, that's outside the scope of the purpose of this lesson, and you can check more information in the additional resources section.



A typical development workflow

Here is a simplified CI workflow that companies often embrace daily as part of their software development process:

- A developer from the team creates a new branch of code in Github, performs changes in the code, and commits them.
- When the developer pushes its work to GitHub, the CI system builds the code on its servers and runs the automated test suite.
- Suppose the CI system detects any error in the CI pipeline. In that case, the developer who pushed the code gets a notification, for example, via email, and the status of CI changes to red. The developer is responsible for analyzing what went wrong and fixing the problem.
- Otherwise, if the status is green and all goes well, the pipeline moves to its next stage, which usually involves deploying a new version of the application to a staging server. This new version can be used internally by the Quality Assurance (QA) team to verify the changes in a production-like environment.

Benefits of continuous integration

Some of the benefits for your software development teams are:

- Improved developer productivity: CI frees developers from manual tasks and the pain of integrating their code with other system parts. They can instead focus on programming the logic that delivers the business's desired features.
- Deliver working software more often: CI is a way for your team to build and test every source code change automatically. This fast CI feedback loop delivers more value to customers than

teams that rely on manual integrations of each other's work. This foundation enables a software delivery process to be efficient, resilient, fast, and secure.

- Find bugs earlier, and fix them faster: The automated testing process can include different checks, like verifying code correctness, validating application behavior, or making sure the coding style follows industry-standard conventions. A CI tool provides instant feedback to developers on whether the new code they wrote works or introduces bugs or regression in quality. Mistakes that are caught early on are the easiest to fix.

Conclusion

In this reading, you have had an introduction to continuous Integration and why embracing a CI tool is important for your software development process.

You also learned about a typical development workflow that can be integrated into your CI system and explored some of the main benefits of using CI.

Completed

Additional resources

Here is a list of additional resources as you continue to explore Integration tests with React Testing Library:

- [React testing library](#) official documentation.
- [Jest](#) official documentation.
- [Continuous delivery](#) is a great article from Atlassian that illustrates the differences between Continuous integration, delivery and deployment, and how they all tie together.
- [Practical test pyramid](#) is an extensive article that dives into the importance of test automation, showing you which kind of tests you should be looking for in the different levels of the pyramid and providing practical examples on how those can be implemented.

Completed

About the final project

What is the purpose of the portfolio project?

The primary purpose of an assessment is to check your knowledge and understanding of the key learning objectives of the course you have just completed. Most importantly, assessments help you

establish which topics you have mastered and which require further focus before completing the course. Ultimately, the assessment is designed to help you make sure that you can apply what you have learned. This assessment's learning objective is to allow you to create a React application or App.

How do I prepare for the portfolio project?

You will have already encountered exercises, knowledge checks, in-video questions and other assessments as you have progressed through the course.

The final project requires you to complete a portfolio in React. You will be provided with code snippets, and your task is to use these, plus any of your own code to complete your developers' portfolio.

This is a creative project, and the goal is to use as many React concepts as possible within this portfolio. You can use component composition, code reusability, hooks, manage state, interact with an external API, create forms, lists and so on.

You will be provided with code snippets and your task is to use these, plus any of your own code, to complete a portfolio app that contains:

- A header with external links to social media accounts and internal links to other sections of the page.
- A landing section with an avatar picture and a short bio.
- A section to display your featured projects as cards in a grid fashion.
- A contact me section with a form to allow visitors to contact you.

Review the final project

You will take part in a peer review exercise in which you will submit your completed portfolio app for two of your peers to review. You will also be required to review two of your peers' portfolio apps.

When you submit your assignment, other learners in the course will review and grade your work.

They will be looking at the Portfolio page functionality based on the following criteria:

- Did the header have external links that take you to different social apps?
- Did the header have internal links that, when clicked, will smoothly scroll into their corresponding section?
- Was the landing section filled with an avatar, name and a short bio?
- Did the project section display a 2x2 grid with each project rendered in a card widget?
- Was the Contact Me form business logic implemented as per the requirements?
- Was the header hidden/shown depending on the scroll direction? Did it happen with a smooth transition animation?
- Can you suggest any improvements for the portfolio app?

You'll also need to give feedback on and grade the assignments of two other learners using the same criteria.

Completed

Popular external libraries

In this reading you will learn about some well-designed UI libraries, such as ChakraUI, that can speed up your application delivery.

You will also explore how to simplify your form design with tools like Formik, and write declarative validation rules with chain operators using Yup.

Chakra UI

UI libraries are a great way to speed up the development process. They provide a set of robust, well-tested and highly configurable pre-built components that you can use to create your applications. Those components act as atoms or building blocks, laying the foundation to create more complex components.

One of the most popular UI solutions is Chakra UI. Chakra UI is a simple, modular and accessible component library that provides you with the building blocks you need for your React applications. Chakra groups its different components by categories, like layout, forms, data display, feedback, typography or overlay.

Layout components are in charge of setting virtual delimiters or boundaries for your content. They also manage how their children are laid (row or column) and the spacing between them among other properties. Some layout components to highlight are:

- HStack and VStack: they display children using flex properties and stack elements horizontally or vertically respectively. Both take a spacing prop that allows you to set the spacing between the elements.
- Box: it allows you to create a box with a background color, border, shadow, etc. It takes a bg prop that allows you to set the background color.

Typography is also an important category that is worth mentioning. There are two main components from this group:

- Heading: renders one of the different DOM header tags (**h1**, **h2**, **h3**...). It takes a size prop that allows you to set the size of the heading and an as prop to specify the particular semantic HTML tag.

1

2

3

```
<Heading as='h2' size='2xl'>
```

```
    Little Lemon
```

```
</Heading>
```

- Text: is used to render text and paragraph within an interface. It offers a **fontSize** prop to increase the font size of the text.

1

```
<Text fontSize='lg'>Best restaurant in town</Text>
```

In order to see all the different component categories and the different props each component accepts, you can check the official [documentation](#) page.

Style props

Chakra uses style props to provide css directives directly as props to the different components. You can find a reference of all the available style props in the [Chakra UI documentation](#).

As a general rule, you can consider *camelCase* versions of css styles to be valid style props. But you can also leverage the shorthand version. For example, instead of using `backgroundColor`, you can use `bg`.

1

```
<Box backgroundColor='tomato' /> is equivalent to <Box bg='tomato' />
```

Putting all together, the below example represents three boxes stacked in a row, with a vertical space of 16px between boxes, where each box has a height of 40px and a different background color, as well as a particular number as its children:

1

2

3

4

5

6

7

8

9

10

11

```
<HStack spacing="16px">
```

```
<Box h='40px' bg='yellow.200'>  
1  
</Box>  
  
<Box h='40px' bg='tomato'>  
2  
</Box>  
  
<Box h='40px' bg='pink.100'>  
3  
</Box>  
  
</HStack>
```

Formik and Yup

Formik is another popular open-source library that helps you to create forms in React. The library takes care of the repetitive tasks of managing the state of the form, validation and submission, so you can focus on the business logic of your application. It does so by providing a set of components and hooks that you can plug into your forms.

Yup is a JavaScript open-source library used to validate the form data before submitting it to the server. It provides a set of chainable operators that you can apply to your form fields to declaratively specify the validation rules.

Formik comes with built-in support for schema based form-level validation through Yup, so they work together seamlessly.

The most important component from Formik is the `useFormik` hook. This hook handles all the different states of your form. It only needs a configuration object as an argument.

Let's break down the hook usage with some code example:

1

2

3

4

5

```
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

import * as Yup from 'yup';

import { useFormik } from 'formik';

// The below code would go inside a React component

const {
```

```

values,
errors,
touched,
getFieldProps,
handleSubmit,
} = useFormik({
initialValues: {
comment: '',
},
onSubmit: (values) => {
// Handle form submission
},
validationSchema: Yup.object({
comment: Yup.string().required("Required"),
}),
});

```

The `useFormik` hook takes an object as an argument with the following properties:

- `initialValues`: An object with the initial values of the form fields
- `onSubmit`: A function that will be called when the form is submitted, with the form values as an argument. In that example you could access the message via `values.comment`.
- `validationSchema`: A Yup schema that will be used to validate the form fields. In that example, the message is a field with a string value that is required. As you can see the rules are human-readable and easy to understand.

The hook returns an object with the following properties:

- **values**: An object with the current values of the form fields. In that example you could access the message via `values.comment`.
- **errors**: An object with the current errors of the form fields. If validation fails for the "comment" field, which would be the case if the input has been touched and its value is empty, according to the validation schema, you could access the message error via `errors.comment`. In that particular case, the message error would be "Required". If the field is valid though, the value will be undefined.
- **touched**: An object with the current touched state of the form fields. You can use this to determine if a field has been touched (focused at least once) or not. In that example, you could access the comment state via `touched.comment`. If the field has been touched, the value will be true, otherwise it will be false.
- **getFieldProps**: A function that takes a field name as an argument and returns an object with the following properties:
 - **name**: The field name.
 - **value**: The current value of the field.
 - **onChange**: The `handleChange` function.
 - **onBlur**: A function that will be called when the field is blurred. It updates the corresponding field in the touched object.

The way you would use this function is by spreading the returned object into the input element. For example, if you had an input element with the name "comment", you would do something like this:

1

```
<input {...getFieldProps("comment")}>
```

- **handleSubmit**: A function that will be called when the form is submitted. It takes an event as an argument and calls the `onSubmit` function with the values object as an argument. You should hook this function to the form `onSubmit` event.

Conclusion

In this reading, you have learned about three of the most popular libraries that can save you precious time during your app development. Their main goal is to take care of mundane and repetitive tasks and let you focus on the stuff that matters.

You were introduced to Chakra UI as a way to leverage well designed components that you can put together to build more complex interfaces.

Finally, you gained knowledge about an open-source library called [Formik](#) and its perfect companion, [Yup](#), to create complex React forms with ease.

Solution code

Here is the completed solution code for the App.js file:

```
1\n\n2\n\n3\n\n4\n\n5\n\n6\n\n7\n\n8\n\n9\n\n10\n\n11\n\n12\n\n13\n\n14\n\n15\n\n16\n\n17\n\n18\n\n19
```

20

21

22

23

24

25

26

27

```
import { ChakraProvider } from "@chakra-ui/react";

import Header from "./components/Header";

import LandingSection from "./components/LandingSection";

import ProjectsSection from "./components/ProjectsSection";

import ContactMeSection from "./components/ContactMeSection";

import Footer from "./components/Footer";

import { AlertProvider } from "./context/alertContext";

import Alert from "./components/Alert";



function App() {

  return (

    <ChakraProvider>

      <AlertProvider>
```

```
<main>

  <Header />

  <LandingSection />

  <ProjectsSection />

  <ContactMeSection />

  <Footer />

  <Alert />

</main>

</AlertProvider>

</ChakraProvider>

);

}

export default App;
```

Here is the completed solution code for the context/alertContext.js file:

1
2
3
4
5
6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

```
import {createContext, useContext, useState} from "react";
```

```
const AlertContext = createContext(undefined);

export const AlertProvider = ({ children }) => {

  const [state, setState] = useState({
    isOpen: false,
    type: 'success',
    message: '',
  });

  return (
    <AlertContext.Provider
      value={ {
        ...state,
        onOpen: (type, message) => setState({ isOpen: true, type, message }),
        onClose: () => setState({ isOpen: false, type: '', message: '' }),
      } }
    >
      {children}
    </AlertContext.Provider>
  );
}
```

```
};  
  
export const useAlertContext = () => useContext(AlertContext);  
  
Here is the completed solution code for the components/Header.js file:  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17
```

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import React, { useEffect, useRef } from "react";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import { faEnvelope } from "@fortawesome/free-solid-svg-icons";
import {

faGithub,
faLinkedin,
faMedium,
faStackOverflow,
} from "@fortawesome/free-brands-svg-icons";
import { Box, HStack } from "@chakra-ui/react";

const socials = [
{
icon: faEnvelope,
url: "mailto: hello@example.com",
},
{
icon: faGithub,
url: "https://www.github.com/sureskills",
}
```

```
    },
    {
      icon: faLinkedin,
      url: "https://www.linkedin.com/in/sureskills/",
    },
    {
      icon: faMedium,
      url: "https://medium.com/@sureskills",
    },
    {
      icon: faStackOverflow,
      url: "https://stackoverflow.com/users/sureskills",
    },
  ],
}

/**
 * This component illustrates the use of both the useRef hook and useEffect
 * hook.
 *
 * The useRef hook is used to create a reference to a DOM element, in order
 * to tweak the header styles and run a transition animation.
 *
 * The useEffect hook is used to perform a subscription when the component
 * is mounted and to unsubscribe when the component is unmounted.
*/
```

* Additionally, it showcases a neat implementation to smoothly navigate to different sections of the page when clicking on the header elements.

*/

Here is the completed solution code for the components/Card.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

```
import { Heading, HStack, Image, Text, VStack } from "@chakra-ui/react";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import { faArrowRight } from "@fortawesome/free-solid-svg-icons";
import React from "react";

const Card = ({ title, description, imageSrc }) => {
```

```
return (
  <VStack
    color="black"
    backgroundColor="white"
    cursor="pointer"
    borderRadius="xl"
  >

  <Image borderRadius="xl" src={imageSrc} alt={title} />

  <VStack spacing={4} p={4} alignItems="flex-start">
    <HStack justifyContent="space-between" alignItems="center">
      <Heading as="h3" size="md">
        {title}
      </Heading>
    </HStack>
    <Text color="#64748b" fontSize="lg">
      {description}
    </Text>
    <HStack spacing={2} alignItems="center">
      <p>See more</p>
      <FontAwesomeIcon icon={faArrowRight} size="1x" />
    </HStack>
  
```

```
</VStack>

</VStack>

);

};

export default Card;
```

Here is the completed solution code for the components/Alert.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

```
import {  
  
  AlertDialog,  
  
  AlertDialogBody,  
  
  AlertDialogContent,  
  
  AlertDialogHeader,  
  
  AlertDialogOverlay,  
  
} from "@chakra-ui/react";  
  
import { useAlertContext } from "../context/alertContext";  
  
import { useRef } from "react";  
  
/**  
  
 * This is a global component that uses context to display a global alert  
 message.  
  
 */  
  
function Alert() {  
  
  const { isOpen, type, message, onClose } = useAlertContext();  
  
  const cancelRef = useRef();  
  
  const isSuccess = type === "success"
```

```

    return (
      <AlertDialog
        isOpen={isOpen}
        leastDestructiveRef={cancelRef}
        onClose={onClose}
      >
      <AlertDialogOverlay>
        <AlertDialogContent py={4} backgroundColor={isSuccess ? '#81C784' : '#FF8A65'}>
          <AlertDialogHeader fontSize="lg" fontWeight="bold">
            {isSuccess ? 'All good!' : 'Oops!'}
          </AlertDialogHeader>
          <AlertDialogBody>{message}</AlertDialogBody>
        </AlertDialogContent>
      </AlertDialogOverlay>
    );
  }

  export default Alert;

```

Here is the completed solution code for the components/Footer.js file:

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

```
import React from "react";
import { Box, Flex } from "@chakra-ui/react";

const Footer = () => {
  return (
    <Box backgroundColor="#18181b">
      <footer>
        <Flex
          margin="0 auto"
          px={12}
          color="white"
          justifyContent="center"
          alignItems="center"
          maxWidth="1024px"
          height={16}
        >
        <p>Pete • © 2022</p>
      </Flex>
    </footer>
  );
}

export default Footer;
```

```
</Box>

);

};

export default Footer;
```

Here is the completed solution code for the components/FullScreenSection.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

16

17

18

19

20

```
import * as React from "react";
import { VStack } from "@chakra-ui/react";

/**
 * Illustrates the use of children prop and spread operator
 */

const FullScreenSection = ({ children, isDarkBackground, ...boxProps }) =>
{
  return (
    <VStack
      backgroundColor={boxProps.backgroundColor}
      color={isDarkBackground ? "white" : "black"}>
      {children}
    </VStack>
  );
}
```

```
</VStack>

);

};

export default FullScreenSection;
```

Here is the completed solution code for the components/LandingSection.js file:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

```
import React from "react";

import { Avatar, Heading, VStack } from "@chakra-ui/react";

import FullScreenSection from "./FullScreenSection";




const greeting = "Hello, I am Pete!";

const bio1 = "A frontend developer";

const bio2 = "specialized in React";




const LandingSection = () => (

<FullScreenSection

justifyContent="center"

alignItems="center"

isDarkBackground

backgroundColor="#2A4365"

>

<VStack spacing={16}>

<VStack spacing={4} alignItems="center">

<Avatar
```

```
        src="https://i.pravatar.cc/150?img=7"
        size="2x1"
        name="Your Name"
    />

    <Heading as="h4" size="md" noOfLines={1}>
        {greeting}
    </Heading>

</VStack>

<VStack spacing={6}>
    <Heading as="h1" size="3x1" noOfLines={1}>
        {bio1}
    </Heading>
    <Heading as="h1" size="3x1" noOfLines={1}>
        {bio2}
    </Heading>
</VStack>

</FullScreenSection>

) ;

export default LandingSection;
```

Here is the completed solution code for the components/ProjectsSection.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

```
import React from "react";
```

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import FullScreenSection from "./FullScreenSection";

import { Box, Heading } from "@chakra-ui/react";

import Card from "./Card";


const projects = [


{



  title: "React Space",



  description:



    "Handy tool belt to create amazing AR components in a React app, with redux integration via middleware",



  getImageSrc: () => require("../images/photo1.jpg"),



},



{


  title: "React Infinite Scroll",



  description:



    "A scrollable bottom sheet with virtualisation support, native animations at 60 FPS and fully implemented in JS land 🔥",



  getImageSrc: () => require("../images/photo2.jpg"),



},



{


  title: "Photo Gallery",



  description:




```

```

    "A One-stop shop for photographers to share and monetize their
photos, allowing them to have a second source of income",
getImageSrc: () => require("../images/photo3.jpg"),
},
{
title: "Event planner",
description:
"A mobile application for leisure seekers to discover unique events
and activities in their city with a few taps",
getImageSrc: () => require("../images/photo4.jpg"),
},
];

```

```

const ProjectsSection = () => {

return (
<FullScreenSection

backgroundColor="#14532d"
isDarkBackground
p={8}
alignItems="flex-start"
spacing={8}

```

Here is the completed solution code for the components/ContactMeSection.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

```
import React, {useEffect} from "react";
import { useFormik } from "formik";
```

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
import {  
  Box,  
  Button,  
  FormControl,  
  FormErrorMessage,  
  FormLabel,  
  Heading,  
  Input,  
  Select,  
  Textarea,  
  VStack,  
} from "@chakra-ui/react";  
  
import * as Yup from 'yup';  
  
import FullScreenSection from "./FullScreenSection";  
  
import useSubmit from "../hooks/useSubmit";  
  
import {useAlertContext} from "../context/alertContext";  
  
/**  
 * Covers a complete form implementation using formik and yup for  
 validation  
 */
```

```

const ContactMeSection = () => {

  const {isLoading, response, submit} = useSubmit();

  const { onOpen } = useAlertContext();

  ...

  const formik = useFormik({
    initialValues: {
      firstName: '',
      email: '',
      type: "hireMe",
      comment: '',
    },
    onSubmit: (values) => {
      submit('https://john.com/contactme', values);
    },
    validationSchema: Yup.object({
      firstName: Yup.string().required("Required"),
      email: Yup.string().email("Invalid email address").required("Required"),
      comment: Yup.string()
    })
  });

  return (
    <Formik
      {...formik}
      render={...}
    >
      ...
    </Formik>
  );
}

export default ContactMeSection;

```

Here is the completed solution code for the hooks/useSubmit.js file:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

```
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

import {useState} from "react";

const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

/**

```

```
* This is a custom hook that can be used to submit a form and simulate an
API call

* It uses Math.random() to simulate a random success or failure, with 50%
chance of each

*/
const useSubmit = () => {

  const [isLoading, setLoading] = useState(false);

  const [response, setResponse] = useState(null);

  const submit = async (url, data) => {

    const random = Math.random();

    setLoading(true);

    try {

      await wait(2000);

      if (random < 0.5) {

        throw new Error("Something went wrong");
      }
    }

    setResponse({
      type: 'success',
      message: `Thanks for your submission ${data.firstName}, we will get
back to you shortly!`,
    });
  }
}
```

```

    } catch (error) {
      setResponse({
        type: 'error',
        message: 'Something went wrong, please try again later!',
      })
    } finally {
      setLoading(false);
    }
  };

  return { isLoading, response, submit };
}

export default useSubmit;

```

In a previous video, you were introduced to a possible solution for the portfolio page, where most of the concepts you learned over the duration of this course were applied in one way or another. However, there are still some interesting extras about the solution that will be illustrated in this reading.

Header animation

In the Header.js component, there are two React core hooks being used: `useRef` and `useEffect`. Those two are used in conjunction to achieve the smooth animation of the header. If you run the application, you can see that the header hides when I am scrolling down, and shows up when I am scrolling back up.

To implement this behavior, I have to use a side effect and subscribe to the scroll event on the window object using `window.addEventListener`.

It's important to remove all subscriptions before the unmounting phase. For that, I have to return a function inside `useEffect` that performs that task. That's the `window.removeEventListener` call you see executed inside that function.

```
1
2
3
4
5
6
7
8
9
10
11

useEffect(() => {
  const handleScroll = () => {
    // Business logic
  };

  window.addEventListener('scroll', handleScroll);

  return () => {
    window.removeEventListener('scroll', handleScroll);
  }
}
```

```
} , [ ]);
```

To animate the header, you need to deal with its underlying DOM node and apply some style transition. Do you recall the React way to do that? If you said `useRef`, you guessed right! That's what I am doing on the container `Box` and `headerRef` holds a reference to the underlying `<div>` node.

```
1
2
3
4
5
6
7
8
9
10
11
12
13

const Header = () => {
  const headerRef = useRef(null);

  ...
  return (
    <Box
      ref={headerRef}
      style={{ position: 'relative' }}>
      <div style={{ height: 100, width: 100, background: 'red' }}></div>
    </Box>
  );
}
```

```
<Box  
    ref={headerRef}  
    {...}  
>  
  
...  
</Box>  
) ;  
};
```

Finally, `handleScroll` is the handler function that will be called every time there is a change in the vertical scroll position.

The meat of this function resides in the comparison between the previous value and the new value. That determines the direction of the scroll and which style I should apply in order to either show or hide the header. Since I am using transition properties in the container `Box` component, the change is animated.

1
2
3
4
5
6
7
8
9
10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
useEffect(() => {  
  
  let prevScrollPos = window.scrollY;  
  
  const handleScroll = () => {  
  
    const currentScrollPos = window.scrollY;  
  
    const headerElement = headerRef.current;  
  
    if (!headerElement) {  
  
      return;  
  
    }  
  
    if (prevScrollPos > currentScrollPos) {  
  
      headerElement.style.transform = "translateY(0)";  
  
    } else {  
  
    }  
  };  
};
```



```
transitionDuration=".3s"

transitionTimingFunction="ease-in-out"

backgroundColor="#18181b"

ref={headerRef}

>

...

</Box>
```

Header navigation

There is another neat trick I would like to show you, which also happens in the Header component. Let's see what happens when I click on one of the header sections. Do you see how it nicely animates and scrolls into its position on the page? Let me show you how simple it is to implement something like that. Coming back to the code, I have this `handleClick` function that is invoked when I click on one of the header navigation items, either Projects or Contact Me.

```
1

2

3

4

5

6

7

8

9

10

const handleClick = (anchor) => () => {
```

```

const id = `${anchor}-section`;

const element = document.getElementById(id);

if (element) {

  element.scrollIntoView({
    behavior: "smooth",
    block: "start",
  });
}

};

}

```

I have defined some `ids` in other sections of the page. For instance, the header of the projects section has an `id` called `project-section`. The `handleClick` function is called with the anchor name depending on where the navigation should happen, as per the code below:

```

1
2
3
4
5
6
7
8

<HStack spacing={8}>

<a href="#projects" onClick={handleClick("projects")}>
  Projects
</a>

```

```
</a>

<a href="#contactme" onClick={handleClick("contactme")}>

  Contact Me

</a>

</HStack>
```

To access that DOM element, you can then use `document.getElementById` and pass the corresponding ID. Once you have it, you can call `element.scrollIntoView` with an object as parameter, setting behavior as smooth and block start. Nice and simple, isn't it?

Formik and Yup validation

[Formik](#) works very nicely with [Yup](#), an open source library that allows you to define validation rules in a declarative way. Let's break down in detail the rules set for the Contact Me form, as part of the `useFormik` hook. `useFormik` hook comes with a `validationSchema` option as part of its configuration object.

1
2
3
4
5
6
7
8
9
10
11

12

13

14

15

16

17

18

```
const formik = useFormik({  
  
  initialValues: {  
  
    firstName: "",  
  
    email: "",  
  
    type: "hireMe",  
  
    comment: "",  
  
  },  
  
  onSubmit: (values) => {  
  
    submit('https://john.com/contactme', values);  
  
  },  
  
  validationSchema: Yup.object({  
  
    firstName: Yup.string().required("Required"),  
  
    email: Yup.string().email("Invalid email  
address").required("Required"),  
  },
```

```
comment: Yup.string()  
  
.min(25, "Must be at least 25 characters")  
  
.required("Required"),  
  
} ,  
  
} );
```

For the `firstName` field, the rule states that it has to be a string and it can't be empty. If empty, Formik will register an error message with the label "Required".

1

```
firstName: Yup.string().required("Required"),
```

The email input is also required. Observe how Yup already provides us with common validators out of the box, like one to verify that what users type is a valid email. If incorrect, Formik will register an error on that input with the error message "Invalid email address". Quite straightforward right?

1

```
email: Yup.string().email("Invalid email address").required("Required"),
```

Finally, I am making the comment field mandatory, with a minimum length of 25 characters.

1

2

3

```
comment: Yup.string()  
  
.min(25, "Must be at least 25 characters")  
  
.required("Required"),
```

Completed

Next steps

Well done on completing this course! You have really enhanced your knowledge and practical ability by completing this course in Advanced React. And now you're one step closer to your goal of becoming a front-end developer. That's something to be proud of! But what might your next steps be? If you choose to continue your journey as a front-end developer, you might want to take the next course in this professional certificate, Principles of UX/UI. In this course you will be introduced to digital User Experience (UX) and User Interface (UI) design on a foundational level. You will learn about the fundamentals of UX and UI design that are core to the success of any website. Each week intends to equip you with the knowledge and skills needed to guide you through the process of UX and UI methodologies to resolve various UX / UI issues.

Or, you can expand your knowledge base and take a course on a related development topic. Just like in this course, you will do a project for the other courses too. This means that you will have another completed project for your growing portfolio which you can use to attract potential employers. That's a smart strategy! Again, congratulations on completing the course. You've built up some good momentum so keep at it!

Completed

