Web Development Crash Course

HTML5 Programming

The Essential Guide to HTML5

Neo D. Truman

Web Development Crash Course

HTML5 Programming

The Essential Guide to HTML5

Neo D. Truman

Copyright

Table of Contents

Preface

In today's world, the Internet is the backbone of almost every business and organization. Therefore, a visually appealing, functional, and user-friendly website is crucial for a successful online presence.

This book is for those who want to learn HTML from scratch and build websites. It is a comprehensive guide covering everything from setting up a development environment to advanced concepts like DOM APIs and Web Worker. The book is divided into 12 chapters, each dealing with a specific aspect of HTML.

Chapter 1 will teach you how to set up a development environment essential for coding HTML. In **Chapter 2**, you will learn the basics of HTML, including the document structure, elements, attributes, favorite icons, comments, and character encoding.

Chapter 3 covers HTML elements, including headings, paragraphs, hyperlinks, lists, tables, iframe, HTML entities and more. **Chapter 4** includes HTML styles and formatting elements like bold, italic, underline, and more. **Chapter 5** focuses on semantic HTML and how to use it to improve the accessibility and search engine optimization of your website.

Chapter 6 will teach you about web forms and their various elements, input types, attributes, and form validation. **Chapter 7** covers HTML multimedia, including images, audio, and video.

In **Chapter 8**, you will learn about Scalable Vector Graphics (SVG) and how to create vector graphics for the web. **Chapter 9** covers HTML Canvas, a powerful tool for creating graphics, animations, and other visual effects on a web page. You will learn to draw text, images, lines, paths and shapes, create gradients, add shadows, and more.

Chapter 10 covers advanced HTML concepts such as block, inline, and inline-block elements. In addition, you will learn how to use data URIs, which allow you to embed images directly into your HTML code.

Chapter 11 covers HTML events, which are actions or occurrences that happen on a web page. You will learn about different events, such as window, form, keyboard, mouse, and clipboard events. You will also learn how to handle these events using JavaScript.

Chapter 12 covers HTML APIs and pre-built JavaScript functions that allow you to add advanced functionality to your web pages. For example, you will learn about the DOM APIs, which would enable you to manipulate HTML elements, and the Geolocation API, which allows you to determine a user's location. You will also learn about the Web Storage API, which allows storing data on a user's device, and the XMLHttpRequest API, enabling you to make HTTP requests from a web page. Last, you will learn about Web Worker, which handles long-running JavaScript code on web pages.

The book includes plenty of examples for learning. By the end of the book, you will have a solid foundation in HTML and be able to create your websites from scratch.

Whether you are a beginner or have some experience with HTML, this book will help you take your skills to the next level. I hope you enjoy reading it and find it helpful in becoming a proficient web developer.

Book Audience

This book is suitable for many readers interested in learning about HTML. It is helpful for beginners who have yet to write any code and experienced developers who want to update their knowledge or learn new techniques. It offers valuable insights and guidance.

Computer science or web development program students can use this book as a textbook or supplementary course material.

Web designers who want to expand their skills beyond visual design and learn how to code their designs using HTML will find this book helpful.

Entrepreneurs who want to build their own websites or web applications but need more technical knowledge can use this book as a guide to get started with HTML.

Small business owners seeking to enhance their website's accessibility and improve search engine optimization can gain valuable insights into the use of semantic HTML from this book.

Developers who want to add advanced functionality to their web pages using HTML APIs and JavaScript can learn how to do so from the later chapters of this book.

Conventions Used in This Book

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Bold

Indicates important concepts or UI items, such as menu items and buttons to be selected or clicked.

Constant width

Indicates computer code, including statements, functions, classes, objects, methods, properties, etc.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or values determined by context.

 This element signifies a general note.


This element signifies a tip or suggestion.


This element indicates a warning or caution.

CHAPTER 1

Setting Up a Development Environment

Setting up a development environment is the first step towards becoming a successful software developer. This chapter will cover the basics of setting up a development environment that will allow you to write and run your code efficiently.

We will begin by installing a code editor, VS Code. Next, we will walk you through installing and configuring the necessary extensions and tools to get started with coding.

This chapter will give you a strong start in programming, whether you are a beginner or an experienced developer who needs to improve your skills. You will learn to set up your development environment and begin your coding journey. So, let's get started!

1.1 Install a Code Editor

The code editor utilized in this book is Visual Studio Code, also known as VS Code. Stack Overflow's survey shows it is the top-rated code editor across different programming languages. To install Visual Studio Code, follow these steps:

- Go to the official Visual Studio Code website at *https://code.visualstudio.com/* and click Download.
- Select the version of Visual Studio Code that corresponds to your operating system.
- After downloading, find and execute the setup file on your computer.
- Follow the installation wizard instructions, including choosing the destination folder and selecting additional options as desired.
- After the installation is complete, launch Visual Studio Code from your applications or programs menu.

1.1.1 Install Essential Extensions

After installing the editor, we can enhance the coding experience with the support of various extensions available in VS Code, helping us to code more efficiently. Here are some essential extensions to add:

- Prettier - Code formatter: Enforces a consistent style by parsing your code and re-printing it with its rules that take the maximum line length into account, wrapping code when necessary.

- Path Intellisense: Visual Studio Code plugin that autocompletes filenames and paths.
- Auto Rename Tag: Auto rename paired HTML/XML tags.
- ESLint: Finds and fixes problems in your JavaScript code.
- vscode-icons: File and folder icons for Visual Studio Code.

* * *

To install an extension in VS Code, follow these steps:

1. Open VS Code: Launch your computer's Visual Studio Code application.

2. Open the Extensions view: Click on the square icon on the left sidebar to open the Extensions view.

3. Search for an extension: You will see a search box at the top in the Extensions view. Enter the extension name you want to install or browse the available extensions.

4. Select an extension: From the search results, click on the extension you want to install. This will open the extension details page.

5. Install the extension: Click the "**Install**" button on the extension details page. The button will change to a progress bar while the extension is downloaded and installed.

1.1.2 Install Live Server

Live Server is a valuable tool enabling developers to create a local development server with a live reload feature for static and dynamic web pages. To install the Live Server extension, click the **Extensions** icon, search for "Live Server", select the extension, and click the **Install** button.

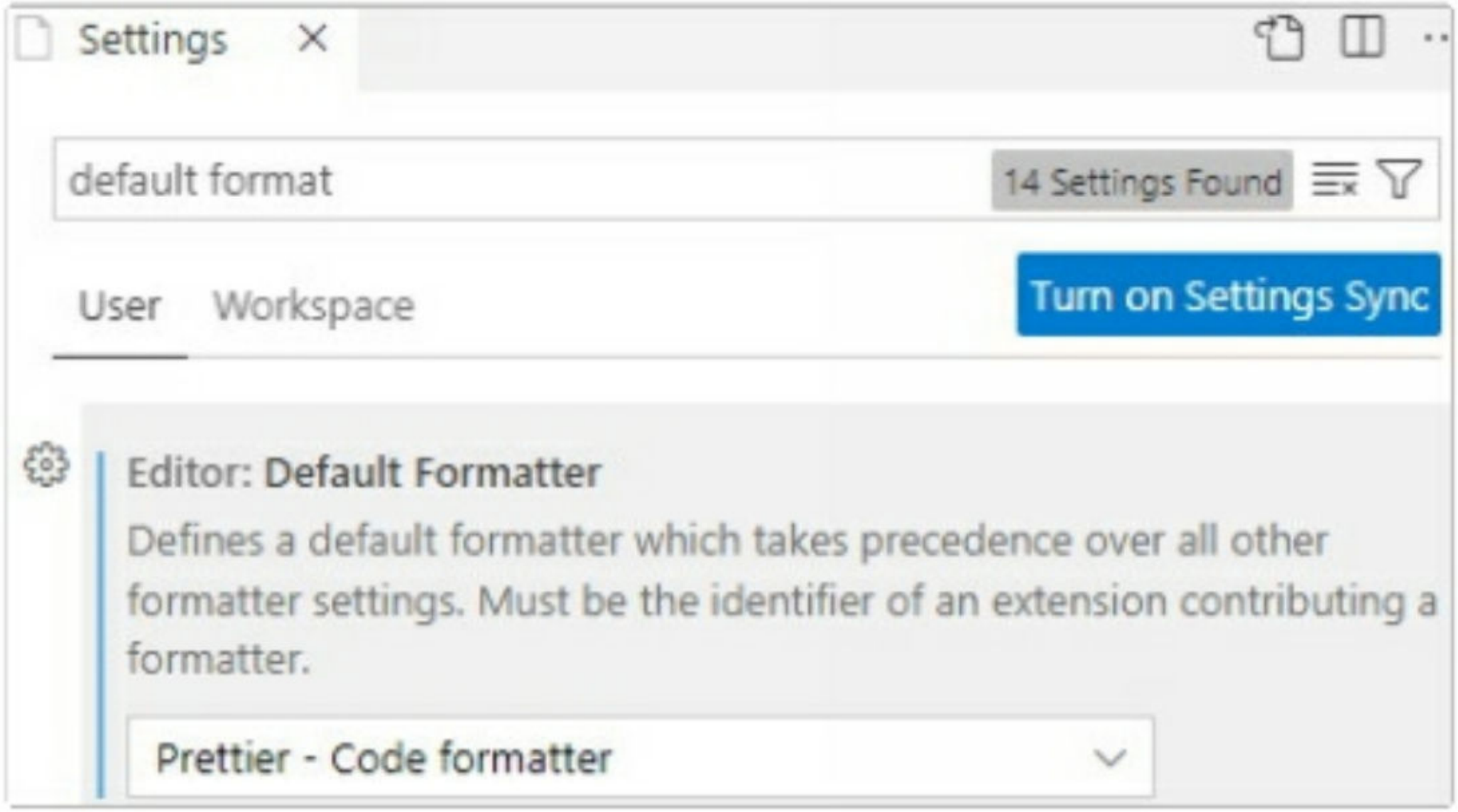
Once installed, you can launch an HTML file with the Live Server by opening it in VS Code, right-clicking on the editing area, and selecting "**Open with Live Server**," or by clicking on the "**Go Live**" button at the bottom-right corner of the VS Code. After completing these steps, your HTML file will be loaded into a live server and available at "*http://127.0.0.1:5500*".

 Please remember to manually refresh your web page once after waking up your PC.

1.1.3 Setting Default Formatter

To use the "**Prettier - Code formatter**" extension as the default formatter, follow these steps:

- Go to the menu and select: **File > Preferences > Settings**
- Type "default format" into the **Search Box**
- Then select the "**Prettier - Code formatter**" option as in the below picture.

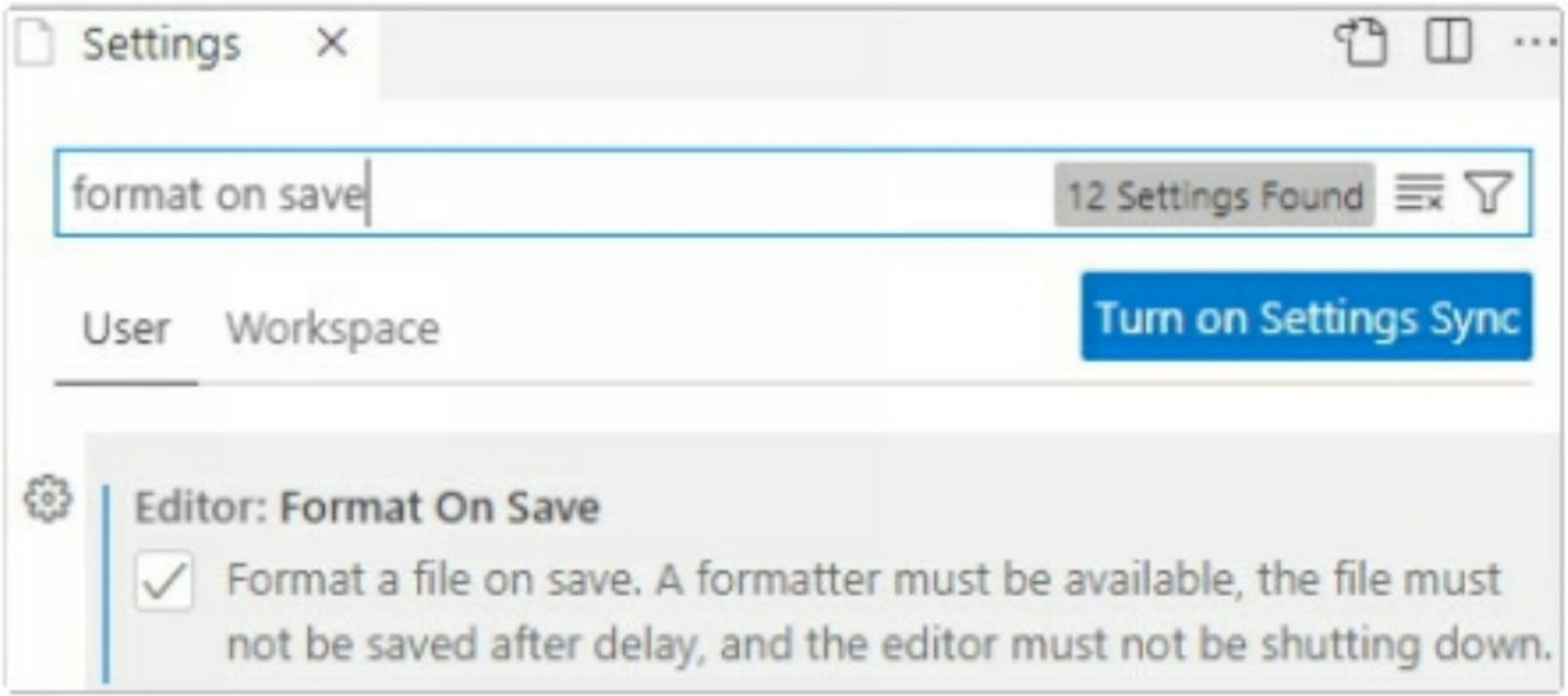


1.1.4 Enable Formatting Code on Save

To ensure the best readability of your code, enabling the "**Format on Save**" feature is recommended. Here's how you can turn it on:

- Go to the menu and select: **File > Preferences > Settings**

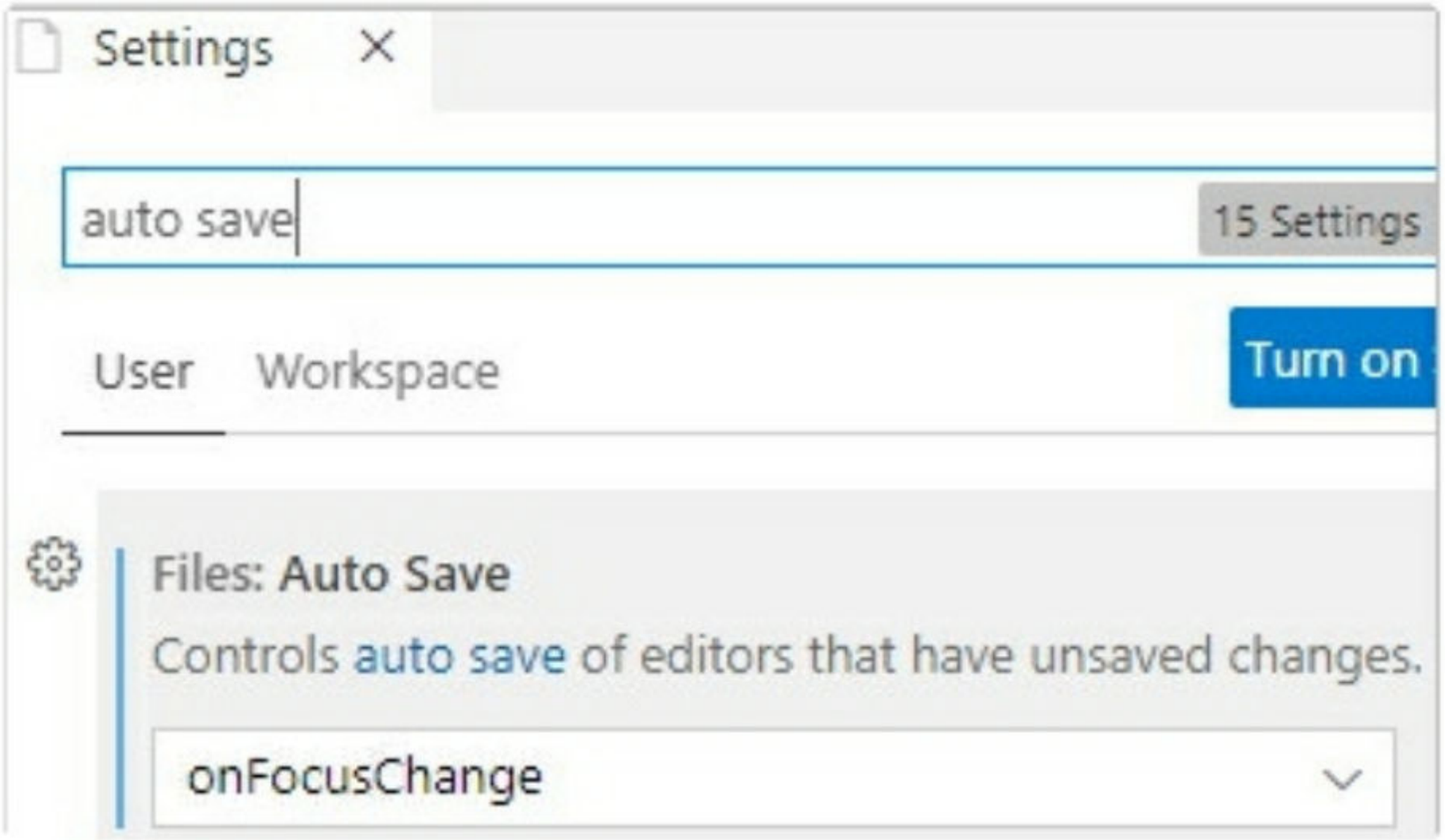
- Type “format on save” into the **Search Box**
- Then check the “**Editor: Format On Save**” checkbox, as shown in the below picture.



1.1.5 Auto Save Files on Focus Change

Enabling the “**File: Auto Save**” feature on focus change can be helpful. Here are the steps to turn it on:

- Go to the menu and select: **File > Preferences > Settings**
- Type “auto save” into the **Search Box**
- Then select the “**onFocusChange**” option as in the below picture.

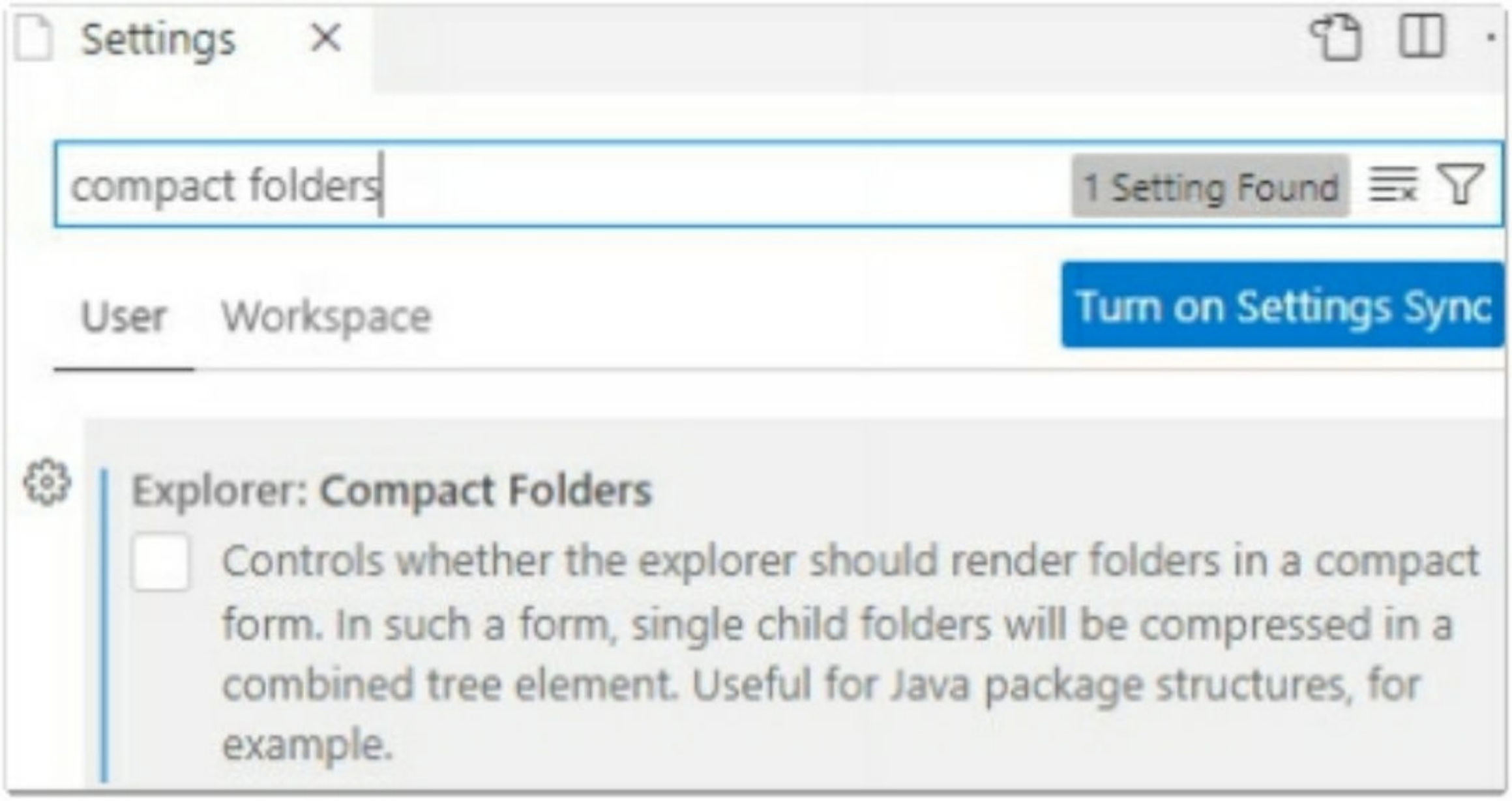


1.1.6 Disable Compact Folders

By default, VS Code displays folders in a compact form in its explorer. It can be helpful for Java package structures but may not be ideal for other projects. To disable the "**Compact Folders**" feature, follow these steps:

- Go to the menu and select: **File > Preferences > Settings**
- Type "compact folders" into the **Search Box**

- Then uncheck the "**Explorer: Compact Folders**" checkbox, as in the picture below.

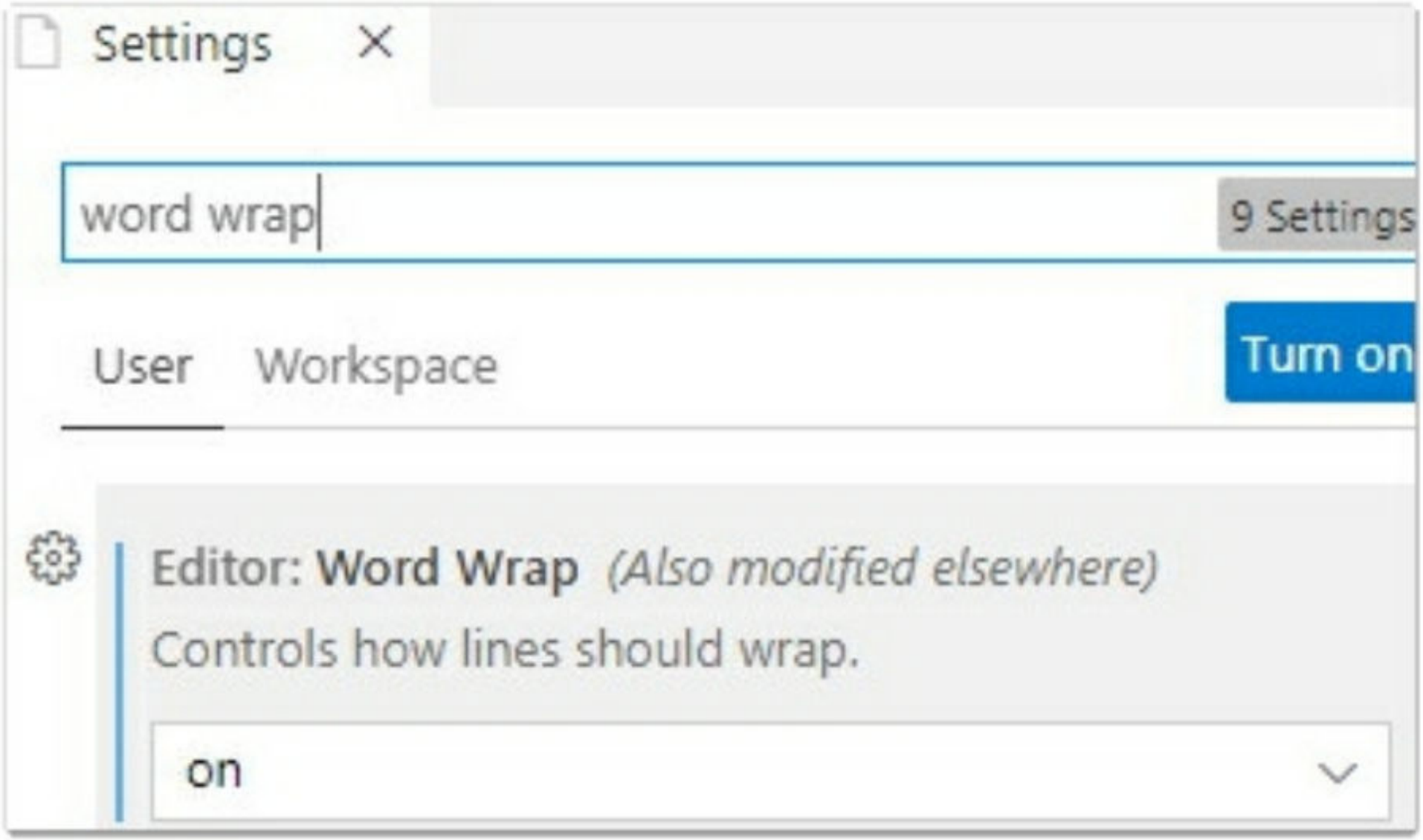


1.1.7 Enable Word Wrap

Enabling the "**Word Wrap**" feature can help eliminate the need for a horizontal scroll bar. Follow these steps to turn it on:

- Go to the menu and select: **File > Preferences > Settings**

- Type “word wrap” into the **Search Box**
- Then select option “**on**” of “**Editor: Word Wrap**” as in the below picture.

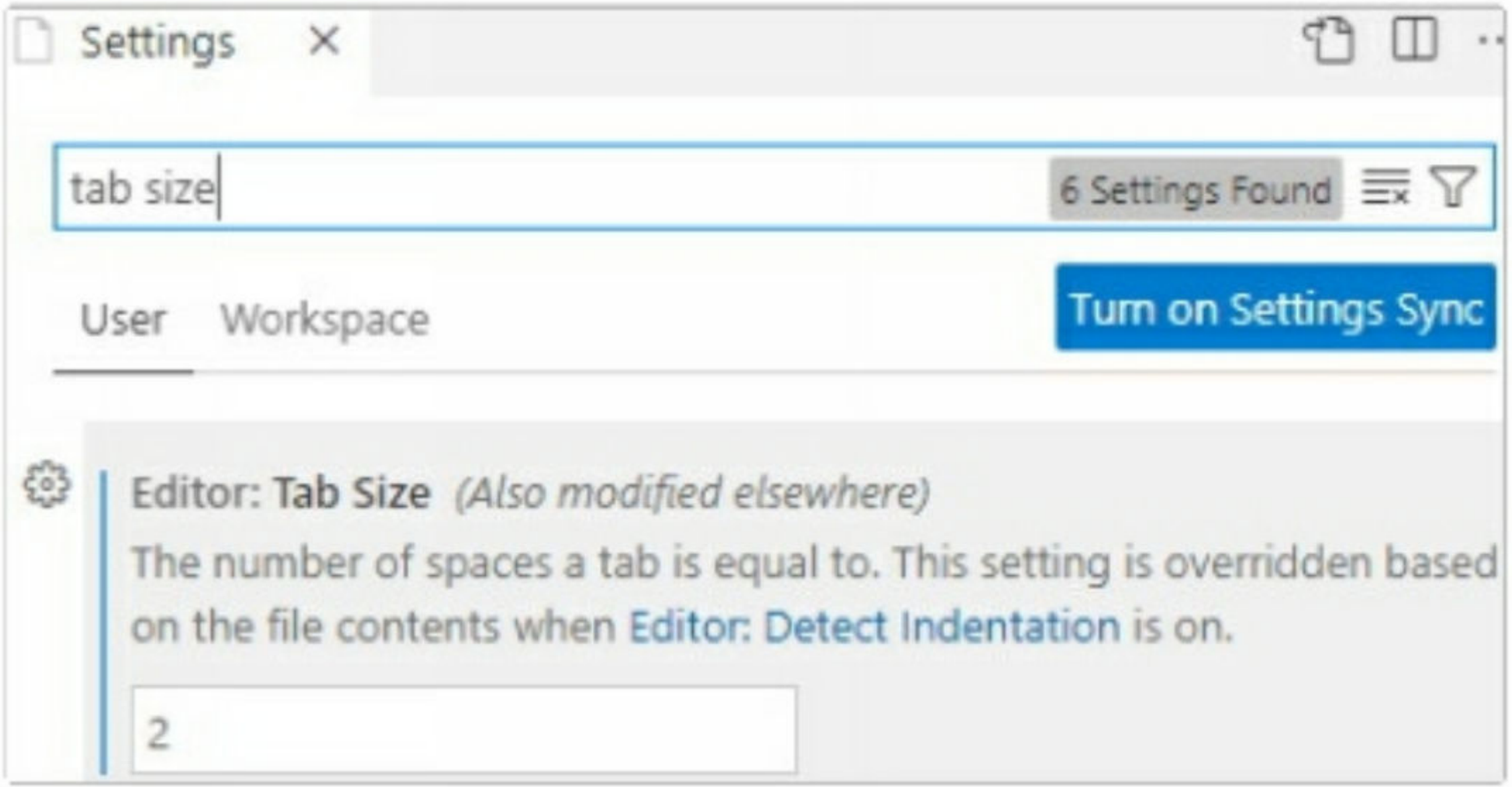


1.1.8 Set Tab Size Smaller

By default, a tab is set to have four spaces. However, based on my experience, we only need two spaces. If you wish to change the default setting to match this, follow these steps:

- Go to the menu and select: **File > Preferences > Settings**

- Type “tab size” into the **Search Box**
- Then set the “**Editor: Tab Size**” number to 2, as shown in the picture below.



1.2 Preparing a Workspace

To create a project in VS Code for learning JavaScript, you can follow these steps:

1. Create a new folder: Create a new folder on your computer where you want to store your project files. You can create it in any location you prefer.

2. Open the project folder: Open VS Code and select "**File**" > "**Open Folder**" to access the project folder created in Step 1. This will open the project folder in VS Code.

3. Create the HTML file: Right-click on the project folder in the VS Code Explorer panel (left sidebar) and choose "**New File**". Name the file "*index.html*" and press Enter.

4. Set up HTML boilerplate: Inside the "*index.html*" file, type the below HTML code.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Welcome to my site!</h1>

    <!-- Your HTML code here -->
  </body>
</html>
```

As you learn from this book, you will modify the HTML code in the "*index.html*" file mentioned above.

5. Run the project: To see your code in action, open the "*index.html*" file in a web browser. Right-click on the "*index.html*" file in the VS Code Explorer panel and choose "**Open with Live Server**" (if you have the Live

Server extension installed) or double-click on the HTML file in the File Explorer to open it with your default browser.

You have created a project in VS Code for learning HTML with the "*index.html*" file. You can now write your HTML code in the file and see the result by opening it in your browser.

CHAPTER 2

Getting to Know HTML

This chapter will dive deeper into HTML, exploring its various elements and document structure. We will begin by discussing how the two types of HTML elements differ. From there, we will move on to HTML document structure and learn about the components of a webpage. You'll also get to create your first web-page and learn about character encoding, element attributes, and how to add a favorite icon to your site. Additionally, we'll cover comments and how they can be used in your code. By the end of this chapter, you will better understand HTML and be ready to move on to more advanced concepts.

2.1 Two Types of HTML Elements

In this section, we will discuss the two types of HTML elements. An HTML element is usually comprised of three parts, as illustrated by the following example:

```
<h1>My First Heading</h1>
```

These parts include:

- The opening tag: <h1>
- The content: “ My First Heading ”
- The closing tag: </h1> . The closing tag is identical to the opening tag but with a forward slash ("/") at the beginning.

However, when an element does not have any content, the closing tag may be omitted, such as in this example:

```
<img src="an-apple.jpg" alt="An apple" />
```

In this case, the image element has no content, so the closing tag is not required.

2.2 HTML Document Structure

To indicate that an HTML document is being used, it is necessary to include a DOCTYPE element at the beginning of the document.

```
<!DOCTYPE html>
```

Next, we create an <html> element, which serves as the root element of the HTML document. This element has two children, as illustrated below.

```
<html>
  <head></head>
  <body></body>
```

```
</html>
```

In all web pages, the structure above remains consistent, with the `<head>` element serving as a container for items that are not visible in the browser. It includes the page title, metadata of the page, links to CSS files, and other related information.

Conversely, visible elements are enclosed within the `<body>` part to ensure they are displayed on the webpage. It is common practice to link JavaScript files at the end of the `<body>` tag.

2.3 Your First Webpage

We can include two additional elements. The initial element is the `<title>` element, which sets the webpage's title and appears in the title tab of the browser. The second element is the `<h1>` element, serving as the main heading of the webpage, and should be used only once.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <h1>Welcome to my site!</h1>
```

```
  </body>
</html>
```

To view the "Welcome to my site!" heading on the page, you can either double-click the "*index.html*" file and open it in a browser or use Live Server.

2.4 Character Encoding

To specify the character encoding for our HTML documents, we can use the " charset " attribute of the <meta> element, which is placed inside the <head> element. The following code shows an example of this in the *index.html* file.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>My Website</title>
  </head>
  <body>
    <h1>Welcome to my site!</h1>
  </body>
</html>
```

The UTF-8 character set is recommended since it covers almost all the characters and symbols in the world. Therefore, it's a good choice for our web pages.

2.5 Element Attributes

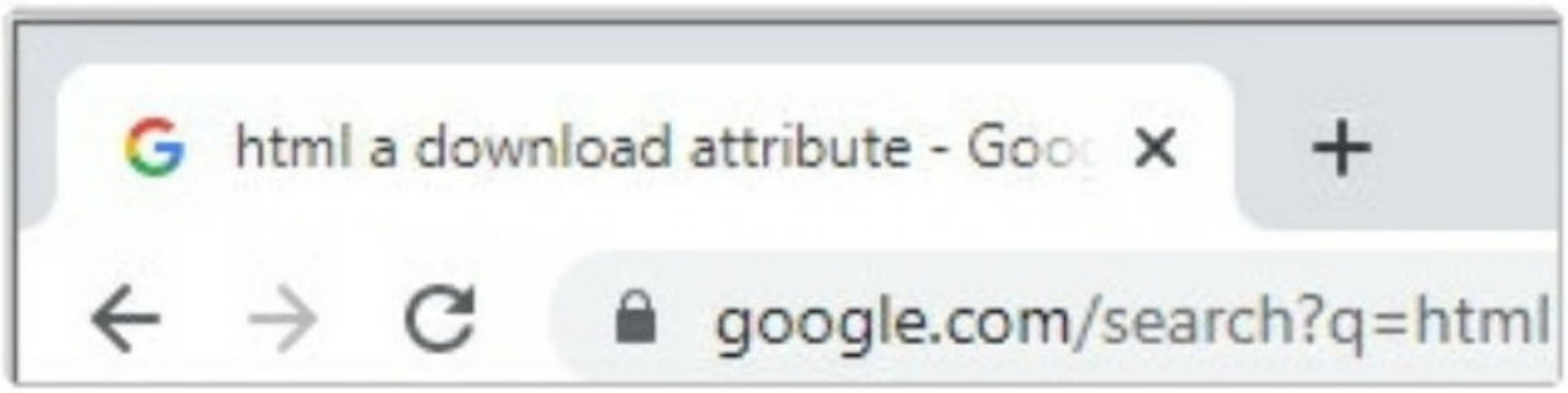
HTML elements have various attributes that can configure the elements, set their contents, or adjust their behavior. For instance, we can use the " lang " attribute to specify the language of our web pages, as shown below:

```
<html lang="en">
```

In the above example, " en " represents "English".

2.6 Favorite Icon

A favorite icon is a small image that appears to the left of the page title in the browser tab, as shown below:



Follow these steps to add a favorite icon to your website:

- Prepare an ICO file as your favorite icon (although the common name is "*favicon.ico*", it's not mandatory).
- Save the ICO file in the root directory or its sub-folder.
- Add a <link> tag to your "*index.html*" file, as shown in the example below.

Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
    <link rel="icon" type="image/x-icon" href="/images/favicon.ico" />
  </head>
  <body>
    <h1>Welcome to my site!</h1>
  </body>
</html>
```

If the favorite icon is located in the root folder, the <link> element's href attribute will change as follows.

```
<link rel="icon" type="image/x-icon" href="/favicon.ico" />
```

2.7 Comments

Comments help explain code and make it more readable. They can also be used to prevent execution when testing alternative code.

We use `<!--` and `-->` to comment out code in HTML. We can comment out code by wrapping it with these symbols, as shown in the example below.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <h1>Heading 1</h1>
    <!-- <h2>Heading 2</h2>
    <h3>Heading 3</h3> -->
  </body>
</html>
```

In the above example, the `<h2>` and `<h3>` headings are commented out, so they won't be displayed on the webpage. Instead, only the `<h1>` header will be visible.

CHAPTER 3

HTML Elements

This chapter covers various HTML elements that are used to structure and display content on a webpage. These elements include headings, paragraphs, hyperlinks, lists, tables, breaks, progress bars, computer code, iframes, and HTML entities. Each element's unique syntax and attributes determine its appearance and behavior on the web pages.

By understanding and effectively using these HTML elements, you can create engaging and visually appealing web pages that deliver information effectively to your audience. This chapter will explore these elements in detail, providing examples and explanations to help you master them.

3.1 Headings

Headings in HTML are used to organize content hierarchically on a web page. HTML provides six levels of headings, ranging from `<h1>` to `<h6>`, with `<h1>` being the most important and having the largest font size, and `<h6>` being the least important and having the smallest font size.

It's vital to use headings correctly for both accessibility and SEO purposes. Screen readers and other assistive technologies use headings to navigate the content of a page, so using them in the correct order and hierarchy can make the page more understandable for disabled people. Search engines also use headings to understand the structure and hierarchy of a page's content, which can affect the page's ranking in search results.

index.html

```
<!DOCTYPE html>
<html>
  <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>
</html>
```

In the example code shown in the "*index.html*" file, all six headings are used to demonstrate their size differences.

3.2 Paragraphs

In HTML, paragraphs are created using the <p> element, which stands for "paragraph". The <p> element groups text content into separate sections on a web page.

When creating a paragraph, it is essential to keep the content concise and easy to read. Therefore, the <p> element should only contain text content and not include other HTML elements such as headings or lists.

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Heading 1</h1>
    <p>This is your first paragraph.</p>
  </body>
</html>
```

In the example provided, the <p> element creates a single paragraph with the text " This is your first paragraph. ". This paragraph will appear below the heading " Heading 1 " on the webpage.

It is worth noting that the <p> element is a block-level element, meaning it will take up the entire width of its container and create a new line after the paragraph content. You can use the element to create inline text within a paragraph.

3.3 Hyperlinks

The hyperlink is a crucial component of HTML. It refers to a webpage or resource that allows users to navigate within a website. Without hyperlinks, visitors would be unable to access other web pages because they wouldn't know their location.

We use the <a> element to create a hyperlink, representing an "anchor." The text displayed on the webpage is determined by the content enclosed between the opening and closing tags of the <a> element.

3.3.1 External Anchors

Users can be directed to another webpage using external anchors, which may belong to the same website or a different one. The " href " attribute specifies the URL the link points to, as shown below.

 Without the “ href ” attribute, an anchor element is not a link.

/index.html

```
<!DOCTYPE html>
<html>
```

```
  <body>
    <h1>Home Page</h1>
    <a href="/contact.html">Contact Page</a>
    <a href="/user/login.html">Login Page</a>
  </body>
</html>
```

/contact.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>This is Contact Page</h1>
    <a href="/">Home Page</a>
  </body>
</html>
```

/user/login.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>This is Login Page</h1>
    <a href="/">Home Page</a>
```

```
  </body>
</html>
```

Users can easily navigate between the "Home", "Contact", and "Login" pages by clicking on the links provided. As "*index.html*" is the primary page, we can use "/" instead of "*/index.html*".

If we want to open an URL in a new tab, we can use the " target " attribute and specify it as " _blank ".

/index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Home Page</h1>
    <a href="/contact.html" target="_blank">Contact Page</a>
    <a href="/user/login.html">Login Page</a>
  </body>
</html>
```

In addition, we can assign the " href " attribute to a URL of a different website, such as "*https://www.google.com*".

3.3.2 Internal Anchors

Internal anchors aid users in navigating within a webpage by utilizing the IDs of its HTML elements. Without an ID, the internal anchors will take the user to the top of the current page. The syntax for this is:

```
<a href="#an-id-here">Description Here</a>
```

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Home Page</h1>
    <a href="#cake">Go to My Cake</a>

    <h2 id="apple">My Apple</h2>
    <img src="an-apple.jpg" alt="This is an apple" height="1000" />

    <h2 id="cake">My Cake</h2>
    <img src="a-cake.jpg" alt="This is a cake" height="1000" />

    <a href="#apple">Go to My Apple</a>
    <a href="#">Back to the top</a>
  </body>
</html>
```

Please prepare two images, "*an-apple.jpg*" and "*a-cake.jpg*", and place them in the same folder where the "*index.html*" file is located. Then, clicking on the "**Go to My Cake**" anchor at the top of the page will bring

the "**My Cake**" heading into view. Similarly, clicking on the "**Back to the top**" anchor will take us to the top of the page.

3.3.3 Download Links

In HTML, the " download " attribute of the anchor tag (<a>) is used to indicate that the target file (specified in the " href " attribute) will be downloaded when clicked by users.

It is possible to assign a new name to the file by setting the value of the " download " attribute, but this is not mandatory. Additionally, if we don't specify the file extension, the browser will automatically detect and append it to the file.

As an example, the following code snippet demonstrates the usage of the " download " attribute in HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <a href="/my-img.jpg" download="new-name">Download Image</a>
  </body>
</html>
```

In this code snippet, when the user clicks on the "**Download Image**" hyperlink, the "*my-img.jpg*" file will be downloaded. However, since we have specified "*new-name*" as the value of the " download " attribute, the downloaded file will be saved with that name instead.

3.4 Lists

3.4.1 Unordered List

Using the <ul> element, which represents an "unordered list", we create a bullet-point list. Each item in the list is made with the <li> element, which stands for "list item", and is nested within the <ul> element, as demonstrated below.

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </body>
</html>
```

The following will be visible on the webpage:

- Item 1

- Item 2
- Item 3

3.4.2 Ordered List

We use the <ol> element to create a numbered list, which represents an "ordered list". Each item in the list is made using the <li> element, which stands for "list item", and is contained within the <ol> element, as shown below.

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <ol>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ol>
  </body>
</html>
```

The following will be visible on the webpage:

1. Item 1

2. Item 2
3. Item 3

3.4.3 Description Lists

We employ a single <dl> element to create a description list. Each description in the list is composed of two parts, utilizing the following elements:

- <dt> - specifies the term or name being defined
- <dd> - provides a description of each term or name

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <dl>
      <dt>HTML</dt>
      <dd>is the standard markup language for Web pages.</dd>
      <dt>CSS</dt>
      <dd>is the language we use to style an HTML document.</dd>
      <dt>JavaScript</dt>
      <dd>is the world's most popular programming language.</dd>
    </dl>
  </body>
```

```
</html>
```

Result:

HTML

: is the standard markup language for Web pages.

CSS

: is the language we use to style an HTML document.

JavaScript

: is the world's most popular programming language.

3.5 Tables

In HTML, the <table> elements are used to display data in rows and columns. They are a helpful tool for presenting information in a structured and organized manner. By defining table rows and columns with HTML tags, we can quickly create tables that can be styled and formatted to suit our needs. Tables can display a wide range of data, from simple lists to complex data sets. For example:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      table {
        border-collapse: collapse;
      }
      th,
      td {
        border: 1px solid black;
      }
      .days {
        background-color: lightyellow;
      }
      .weekend {
```

```
        background-color: lightblue;
      }
    </style>
  </head>
  <body>
    <table>
      <caption>
        September 2022
      </caption>
      <colgroup>
        <col span="5" class="days" />
        <col span="2" class="weekend" />
      </colgroup>
      <thead>
        <tr>
          <th>Mon</th>
          <th>Tue</th>
          <th>Wed</th>
          <th>Thu</th>
          <th>Fri</th>
          <th>Sat</th>
          <th>Sun</th>
```

```
    </tr>
  </thead>
  <tbody>
    <tr>
      <td></td>
      <td></td>
      <td></td>
      <td>1</td>
      <td>2</td>
      <td>3</td>
      <td>4</td>
    </tr>
    <tr>
      <td>5</td>
      <td>6</td>
      <td>7</td>
      <td>8</td>
      <td>9</td>
      <td>10</td>
      <td>11</td>
    </tr>
    <tr>
```

```
    <td>12</td>
    <td>13</td>
    <td>14</td>
    <td>15</td>
    <td>16</td>
    <td>17</td>
    <td>18</td>
  </tr>
  <tr>
    <td>19</td>
    <td>20</td>
    <td>21</td>
    <td>22</td>
    <td>23</td>
    <td>24</td>
    <td>25</td>
  </tr>
  <tr>
    <td>26</td>
    <td>27</td>
    <td>28</td>
    <td>29</td>
```

```
            <td>30</td>
            <td colspan="2"></td>
          </tr>
        </tbody>
      </table>
    </body>
</html>
```

This is an example of an HTML table displaying a calendar for September 2022. The <table> element is used to create the table, with various child elements defining the structure and content of the table.

The <colgroup> element specifies the number of columns in the table and assigns specific classes to each column to be styled with CSS.

The <thead> and <tbody> elements define the table header and body, respectively, while <th> and <td> elements are used to create table cells containing column headers and data, respectively.

The table is also styled using CSS, with a " border-collapse " property applied to the table element to remove borders between table cells. The " days " and " weekend " classes are assigned to specific columns in the table, applying background colors to the cells in those columns. Finally, the " colspan " attribute is used to span two cells in the final row of the table.

Result:

3.5.1 Table Headers

Table headers are not required, but in many cases, we place them inside a <thead> element. In the example, the table header comprises <th> elements within a <tr> element.

3.5.2 Table Rows

In the example, a table row comprises <td> elements and is enclosed within a <tr> element. Including this <tr> element within a <tbody> element is standard practice.

3.5.3 Table Caption

To provide a caption for a table, we can use the <caption> element, typically added directly after the

<table> tag. By default, the caption will be positioned above the table and centered.

3.5.4 Group of Columns

To format specific groups of columns within a table, we can use the <colgroup> element. This element includes one or more <col> elements, with each <col> element representing a group of columns. We can utilize the " span " attribute within the <col> element to group columns together. Additionally, we can use the " class " attribute to apply CSS styling to table cells based on class names.

3.5.5 Column Span and Row Span

The <td> element's " colspan " attribute determines the number of columns a cell should occupy within a table. The <td> element also includes a " rowspan " attribute used to merge cells vertically.

3.6 Breaking

3.6.1 Line Breaking

In HTML, line breaking refers to how text is displayed on a webpage, including when and where a text should break and continue on the following line. Proper line breaking is essential for the readability and aesthetics of a webpage and for ensuring that content is displayed correctly across different devices and screen sizes.

To add a line break on a webpage, you can use the <br> element. Here's an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>I've added a line break here.<br />This sentence will be on the second line.</p>
  </body>
</html>
```

The text inside the paragraph says, " I've added a line break here. " The next part of the text contains a line break created with the `<br>` tag. The `<br>` tag is self-closing, which means it does not require a matching closing tag. After the line break, " This sentence will be on the second line. " is added.

3.6.2 Thematic Breaking

When designing a webpage, ensuring the content is well-organized and easy to navigate is essential. One way to do this is using thematic breaks to separate different page sections visually.

The `<hr>` element in HTML is commonly used to create a horizontal line that divides content. This simple yet effective tool can help to make a webpage more readable and user-friendly by guiding the user's eye and signaling topic shifts. In this section, we will look at an example of how it can enhance your web-page's design.

index.html

```
<!DOCTYPE html>
```

```
<html>
  <body>
    <h1>Web Development</h1>
    <p>HTML is the standard markup language for web pages.</p>
    <hr />
    <p>CSS is the language we use to style an HTML document.</p>
    <hr />
    <p>JavaScript is the world's most popular programming language.</p>
  </body>
</html>
```

3.7 Progress Bars

3.7.1 <progress>

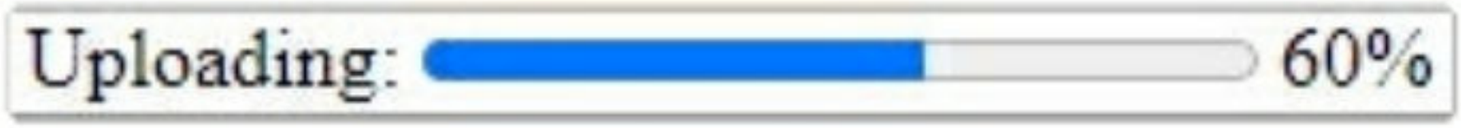
The <progress> tag allows us to display the progress of a task being completed. The " value " attribute indicates the current progress and the " max " attribute is typically set to 100. Here is an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>
```

```
        Uploading:
        <progress value="60" max="100"></progress>
        60%
      </p>
    </body>
</html>
```

Result:



3.7.2 <meter>

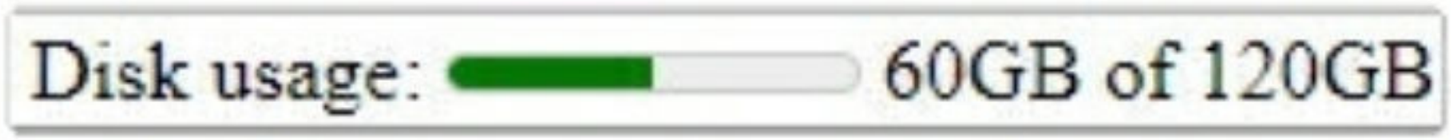
The <meter> tag represents a measurement within a specific range, like the usage of a dish. In this HTML tag, the " value " attribute defines the current value of the meter, while the " max " attribute sets the maximum limit that the meter can reach. Here's an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>
      Disk usage:
```

```
        <meter value="60" max="120"></meter>
        60GB of 120GB
      </p>
    </body>
</html>
```

Result:



3.8 Computer Code

3.8.1 <code>

The <code> tag indicates computer code within a webpage. The text within the tag will be displayed in the browser's default monospace font. Here's an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>This is a piece of JavaScript code:</p>
    <code>var x = 10;</code>
```

```
  </body>
</html>
```

3.8.2 <pre>

When text is placed inside a <pre> tag, it will be displayed in the browser's default monospace font. The text will be displayed as it is written in the HTML code, including any spaces and line breaks. Here's an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <pre>
      Text in a pre element is displayed
      in the browser's default monospace font,
      and it preserves both     spaces
      and line breaks
    </pre>
  </body>
</html>
```

3.9 Iframes

When creating a webpage, you can embed content from another website or a web page within your own. This is where the <iframe> tag comes in handy. An iframe, short for "inline frame", allows you to display a web page within a frame on your page.

This can be useful for showing external content, such as maps, videos, or social media feeds while keeping users on your site. In this section, we will explore the basics of using iframes in HTML and look at an example of how they can enhance your webpage's functionality. Here's an example:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      #my-iframe {
        width: 800px;
        height: 600px;
        border: none;
      }
    </style>
  </head>
```

```
  <body>
    <h1>Web Development Books</h1>
    <iframe id="my-iframe" src="https://www.amazon.com/dp/B09VFLS7TF"></iframe>
  </body>
</html>
```

This code creates an iframe element with an id of " my-iframe ". The URL of the external webpage to be displayed within the iframe is specified by the " src " attribute. In this case, the URL is *https://www.amazon.com/dp/B09VFLS7TF*, which is the link to my product page.

When this code is rendered in a browser, an iframe element will be displayed on the webpage. The iframe's contents will be the Amazon product page specified by the " src " attribute. In addition, the id " my-iframe " can help manipulate or style the iframe using JavaScript or CSS.

An <iframe> element is displayed with a border by default. However, we can use CSS and the " border " property to remove this border, as demonstrated in the example above.

3.10 HTML Entities

HTML entities are special characters that have a specific meaning in HTML code. These entities are used to represent characters that may be reserved for HTML markups, such as the less than sign (<) or the ampersand symbol (&), or characters that may not be easily typed on a keyboard, such as accented letters or mathematical symbols.

HTML entities ensure that web browsers correctly display the intended characters in web pages, regardless of the user's operating system or browser. They are essential for creating accessible and inclusive web content that a wide range of users can understand. In this sense, understanding HTML entities is fundamental for any web developer or content creator.

The syntax of an HTML entity looks like this:

```
&entity-name;
```

or

```
&#entity-number;
```

These are some popular HTML entities:

Result	Description	Entity Name	Entity Number
	non-breaking space		
<	less than	&lt;	&#60;
>	greater than	&gt;	&#62;
&	ampersand	&amp;	&#38;

¢	cent	¢	¢
£	pound	£	£
¥	yen	¥	¥
€	euro	€	€
©	copyright	©	©
®	registered trademark	®	®
™	trademark	™	™
←	leftwards arrow	←	←
↑	upwards arrow	↑	↑
→	rightwards arrow	→	→
↓	downwards	↓	↓

	arrow		

Below is an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <footer>&copy; 2022, Neo D. Truman</footer>
  </body>
</html>
```

CHAPTER 4

HTML Styles

In web development, HTML is the foundation of every web page, providing structure and content to the website. However, more than HTML alone is needed to enhance a webpage's visual appearance fully. This is where CSS comes in handy, allowing developers to add styles and formatting to HTML elements to create a visually appealing and engaging web page.

In this chapter, we will delve into the various ways of adding styles to HTML elements through formatting elements such as <b>, <i>, <u>, <s>, <mark>, <sub> and <sup> , and through the HTML " style " attribute. By the end of this chapter, you will understand how to add style and formatting to your web pages using HTML.

4.1 Formatting Elements

4.1.1 <b>

One way to make text appear bold is by enclosing it inside a <b> HTML element. The letter "b" in this

tag stands for "bold". For instance:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>This is your <b>first paragraph</b></p>
  </body>
</html>
```

In the above example, the text "**first paragraph**" will appear bold when reloads the webpage.

4.1.2 <i>

In addition, we can use the <i> HTML element to render text in italic. The letter "i" in this tag stands for "italic". For instance:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>This is your <i>first paragraph</i></p>
  </body>
</html>
```

In the above example, the text "*first paragraph*" will appear in italic when the webpage is loaded.

4.1.3 <u> or <ins>

The <u> or <ins> HTML element can display text with an underline. For example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>This is some <u>mispeled</u> text.</p>
  </body>
</html>
```

In the above example, the word "mispeled" will be underlined when the webpage is loaded.

4.1.4 <s> or <del>

The <s> or <del> HTML element can be used to display inaccurate text, and will be shown with a line through it. For example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
```

```
    <p><s>These are</s>This is some text.</p>
  </body>
</html>
```

In the above example, the words "~~These are~~" will have a line through them, indicating that they are inaccurate, while "This is some text." will be displayed normally when the webpage is loaded.

4.1.5 <mark>

The <mark> HTML element highlights text by changing its background color. For instance:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>This is an <mark>important</mark> paragraph</p>
  </body>
</html>
```

In the above example, the word "important" will be highlighted with a different background color when the webpage is loaded.

4.1.6 <sub> and <sup>

The <sub> HTML element displays subscript text, which appears half a character below the regular

line and is rendered in a smaller font. An example of subscript text is the "1" in "X_1".

The <sup> HTML element shows superscript text slightly above the regular line and is smaller in font size. An example of superscript text is the "2" in "X^2".

Here is an example usage of both <sub> and <sup> elements in an HTML document:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>This is a <sub>subscript</sub> text.</p>
    <p>This is a <sup>superscript</sup> text.</p>
  </body>
</html>
```

4.2 HTML Style Attribute

HTML elements can have CSS styles applied to them using the “ style ” attribute. This inline styling will only affect the element with the “ style ” attribute. In the following example, the first paragraph has inline styles applied to it, but not the second paragraph:

index.html

```
<!DOCTYPE html>
```

```
<html>
  <body>
    <p style="color: red; font-size: 26px">This is the first paragraph.</p>
    <p>This is the second paragraph.</p>
  </body>
</html>
```

However, it is generally recommended to avoid using inline CSS for the following reasons:

- It can clutter the HTML code and make it harder to read and understand.
- It can make maintenance and updating of the code more difficult.

CHAPTER 5

Semantic HTML

Semantic HTML is a fundamental concept in web development that emphasizes the importance of using HTML elements that convey meaning and structure to the browser and the developer. Although they carry special meanings, these HTML elements do not have any distinctive appearance when rendered in a web browser.

In this chapter, we will explore the significance of semantic HTML and how it improves accessibility, search engine optimization, and overall code readability. We will also learn about the various semantic HTML elements we can use to create well-structured and accessible web pages. By the end of this chapter, you will better understand how to use semantic HTML to enhance the quality and usability of your web pages.

5.1 Semantic Elements

5.1.1 <blockquote>

The <blockquote> element is used to display a quoted portion from another source, with the URL of the source defined using the " cite " attribute. For instance, you can use this element to showcase a quote from a book, website, or article. Here is an example of using <blockquote> in HTML:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>Here is a quote from Wikipedia:</p>
    <blockquote cite="https://en.wikipedia.org/wiki/HTML">The HyperText Markup Language or HTML is the standard markup language for documents designed to be displayed in a web browser.</blockquote>
  </body>
</html>
```

Additionally, the <q> tag can be used to define a short quotation. Although not a semantic element, it can add quotes to text. Here is an example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Wikipedia's goal is to: <q>Provide a free encyclopedia that anyone can edit.</q></p>
  </body>
```

```
</html>
```

5.1.2 <address>

The <address> element is used to specify the contact details of the author of an article or document in HTML. The contact details can include the author's name, address, phone number, and email address. In the following example, the <address> element provides the author's contact details.

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <address>
      <p>12345 South St. Philadelphia, PA 12345</p>
      <p>
        <a href="tel:123-456-7890">123-456-7890</a><br />
        <a href="mailto:contact@neodtruman.com">contact@neodtruman.com</a>
      </p>
    </address>
  </body>
</html>
```

5.1.3 <details> and <summary>

The <details> element helps create expandable content that users can open and close. By default, it is collapsed and shows only the summary or heading.

The <summary> element is a child of <details> and defines the header for the details.

Here is an example of how these elements can be used to create an expandable section about HTML on a web page:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <details>
      <summary>HTML</summary>
      <p>HTML is the standard markup language for Web pages.</p>
    </details>
  </body>
</html>
```

5.1.4 <figure> and <figcaption>

The <figure> element is used to add images to HTML documents, and we can use the <figcaption>

element to provide a caption for it. In addition, the <figcaption> element can provide additional context or information about the image. Here's an example of how to use these elements:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <figure>
      <img src="an-apple.jpg" alt="An apple" />
      <figcaption>Fig.1 - My apple</figcaption>
    </figure>
  </body>
</html>
```

5.2 Semantic Layout Elements

HTML provides a range of semantic elements allowing developers to specify different web page parts with meaningful and descriptive names. These elements include:

- <article> element defines independent content.
- <main> element specifies the main content of the HTML document.
- <section> element defines a section in an HTML document.

- <header> element represents the top part of a web document or the top part of some minor elements like <section> and <article> .
- <footer> element is used to wrap content that comes at the end of a page, such as the copyright or at the end of <section> and <article> .
- <aside> element represents a section of a page that includes content related to the main content.
- <nav> stands for "navigation," and it is where developers typically place navigation links.

Example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <header>
      <nav>
        <a href="#">Logo</a>
        <ul>
          <li><a href="#">Posts</a></li>
          <li><a href="#">Contact</a></li>
          <li><a href="#">Login</a></li>
        </ul>
      </nav>
```

```
    <h1>Main Heading</h1>
  </header>

  <main>
    <section>
      <h2>Section's Heading 1</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut eget ex eget lacus luctus porta ac ut nisi. Nunc euismod maximus faucibus. In hac habitasse platea dictumst. Sed molestie consequat massa, non sollicitudin metus dapibus eu.</p>
    </section>

    <section>
      <h2>Section's Heading 2</h2>
      <p>In egestas dictum suscipit. Vestibulum condimentum tristique imperdiet. Quisque lacinia eros et tempus volutpat. Morbi malesuada eleifend sapien, id tristique urna porta maximus.</p>
    </section>
  </main>

  <footer>
    <p>Copyright &copy; 2022 by Neo D. Truman</p>
    <address>
      <p>12345 South St. Philadelphia, PA 12345</p>
```

```
        <p>
          <a href="tel:123-456-7890">123-456-7890</a><br />
          <a href="mailto:contact@neodtruman.com">contact@neodtruman.com</a>
        </p>
      </address>
    </footer>
  </body>
</html>
```

This HTML document has a header, main content section, and footer. The header includes a navigation menu with links to "Posts", "Contact", and "Login", as well as a main heading "Main Heading". The main content section includes two sections with their respective headings and paragraphs of text. Finally, the footer contains copyright information and the website owner's contact information.

This HTML code demonstrates the use of semantic HTML elements such as `<header>`, `<nav>`, `<section>` , and `<footer>` to structure a webpage for better accessibility and search engine optimization.

5.3 Semantic Formatting Elements

5.3.1

Using the `<strong>` element instead of the `<b>` tag is recommended, as it carries semantic meaning. The `<strong>` element signifies important content that requires distinction from other content on the

page. For instance, in the following example, the tag is used to highlight a specific phrase in a paragraph:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Heading 1</h1>
    <p>This is your <strong>first paragraph</strong></p>
  </body>
</html>
```

5.3.2

The tag is used to provide emphasis on a particular content, where "em" stands for "emphasize". For instance:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Heading 1</h1>
    <p>This is your <em>first paragraph</em></p>
```

```
    </body>
</html>
```

CHAPTER 6

Web Forms

Online forms have become essential to everyday life in this digital age. We encounter online forms daily, from signing up for a newsletter to submitting a job application. HTML web forms are the backbone of these online forms, providing a platform for users to enter and submit data to the website's backend.

This chapter will explore the various elements and attributes that makeup HTML forms, including input types such as text, radio buttons, checkboxes, and dropdown menus. We will also cover how to add labels, placeholders, and validation to ensure the user enters the correct data. By the end of this chapter, you will have a comprehensive understanding of creating and customizing HTML forms, allowing you to create interactive and engaging web pages.

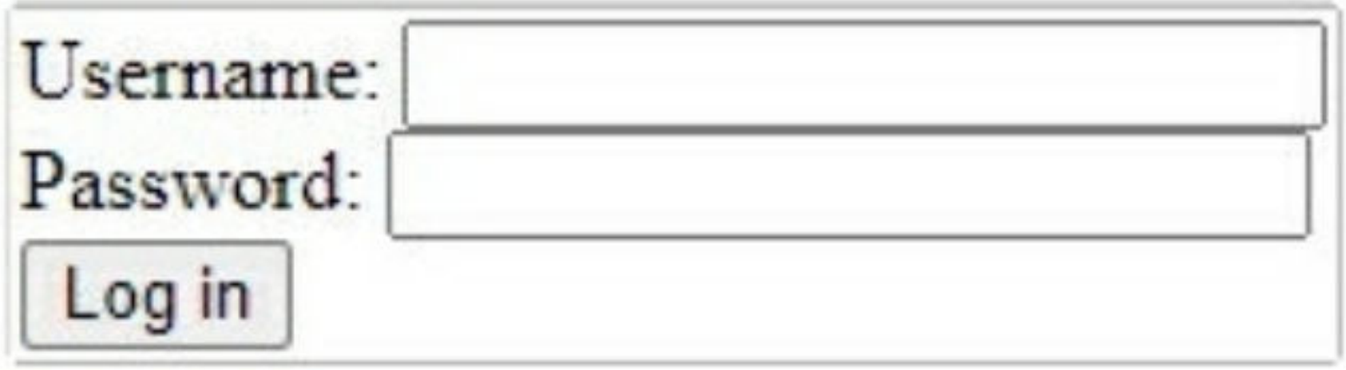
We often encounter a simple form like the one below on web pages.

index.html

```
<!DOCTYPE html>
<html>
  <body>
```

```
    <form action="/login">
      <label for="acc-name">Username:</label>
      <input type="text" id="acc-name" name="username" /><br />
      <label for="acc-pass">Password:</label>
      <input type="password" id="acc-pass" name="password" /><br />
      <input type="submit" value="Log in" />
    </form>
  </body>
</html>
```

Result:



Once you input your desired username and password in the respective fields of the form, clicking the "**Log in**" button will redirect you to the "*/login*" page. For example, suppose you entered " john " as your username and " 123 " as your password. In this case, the resulting URL would be:

http://127.0.0.1:5500/login?username=john&password=123

It indicates that the browser has initiated an HTTP request to the webserver to retrieve the webpage located at the address "*http://127.0.0.1:5500/login*", which includes a query string containing the entered username and password. The " name " attributes of the <input> elements correspond to " username " and " password " in the query string. The server will use this information to authenticate the user.

6.1 HTML Form Elements

6.1.1 <input> and <label>

The <input> tag is utilized to gather user input, with various input types discussed in the upcoming section. Additionally, a <label> tag is commonly used to guide the data the user should enter into the <input> tag. The " for " attribute of the <label> tag identifies the corresponding form element using its ID, allowing the input to receive focus when the label is clicked.

Example:

```
<label for="fname">First name:</label>
<input type="text" id="fname" name="firstname" />
```

Result:

6.1.2 <textarea>

The <textarea> tag creates a text input control that allows users to input multiple lines of text. For instance:

```
<textarea name="comment" rows="3" cols="30"> </textarea>
```

Result:

As a result, users can enter an unlimited number of lines of text into the <textarea> tag. Furthermore, they can adjust the input size by dragging the bottom-right corner of the <textarea> element.

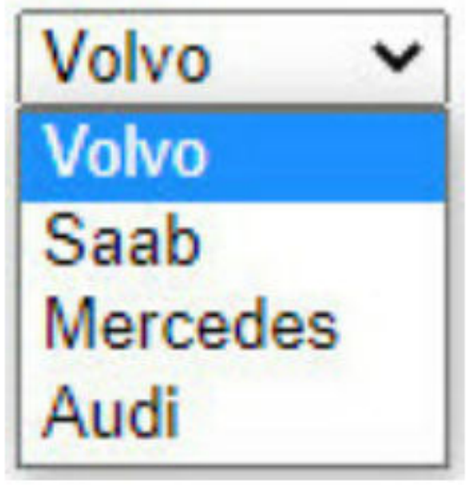
6.1.3 <select> and <option>

The <select> tag defines a drop-down list containing numerous options, each defined by an <option> tag. For example:

```
<select name="car">
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
```

```
  <option value="audi">Audi</option>
</select>
```

Result:

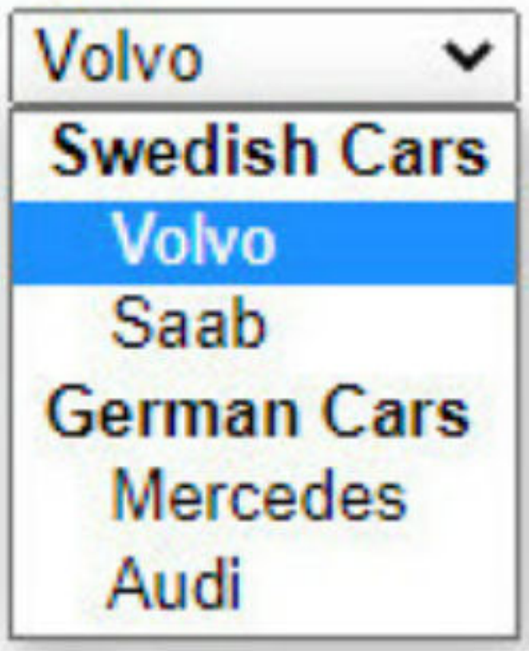


In a <select> tag, we can use the <optgroup> tag to group related options together. The "label" attribute of the <optgroup> tag is shown in the drop-down list to differentiate the various groups, which makes it easier to understand. Here is an example:

```
<select name="car">
  <optgroup label="Swedish Cars">
    <option value="volvo">Volvo</option>
    <option value="saab">Saab</option>
  </optgroup>
  <optgroup label="German Cars">
    <option value="mercedes">Mercedes</option>
    <option value="audi">Audi</option>
```

```
  </optgroup>
</select>
```

Result:

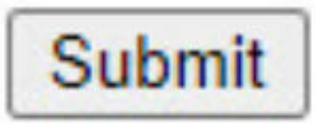


6.1.4 <button>

The <button> tag creates a clickable button, with the button's text placed between the opening and closing tags. If a user clicks on a <button> element inside a <form> , all of the form's values will be sent to the server. Here is an example:

```
<button>Submit</button>
```

Result:



While optional, the " onclick " attribute can be used to specify which JavaScript code should be executed when a user clicks on the button. Once the JS code has been executed, the form values will be submitted to the server. For instance:

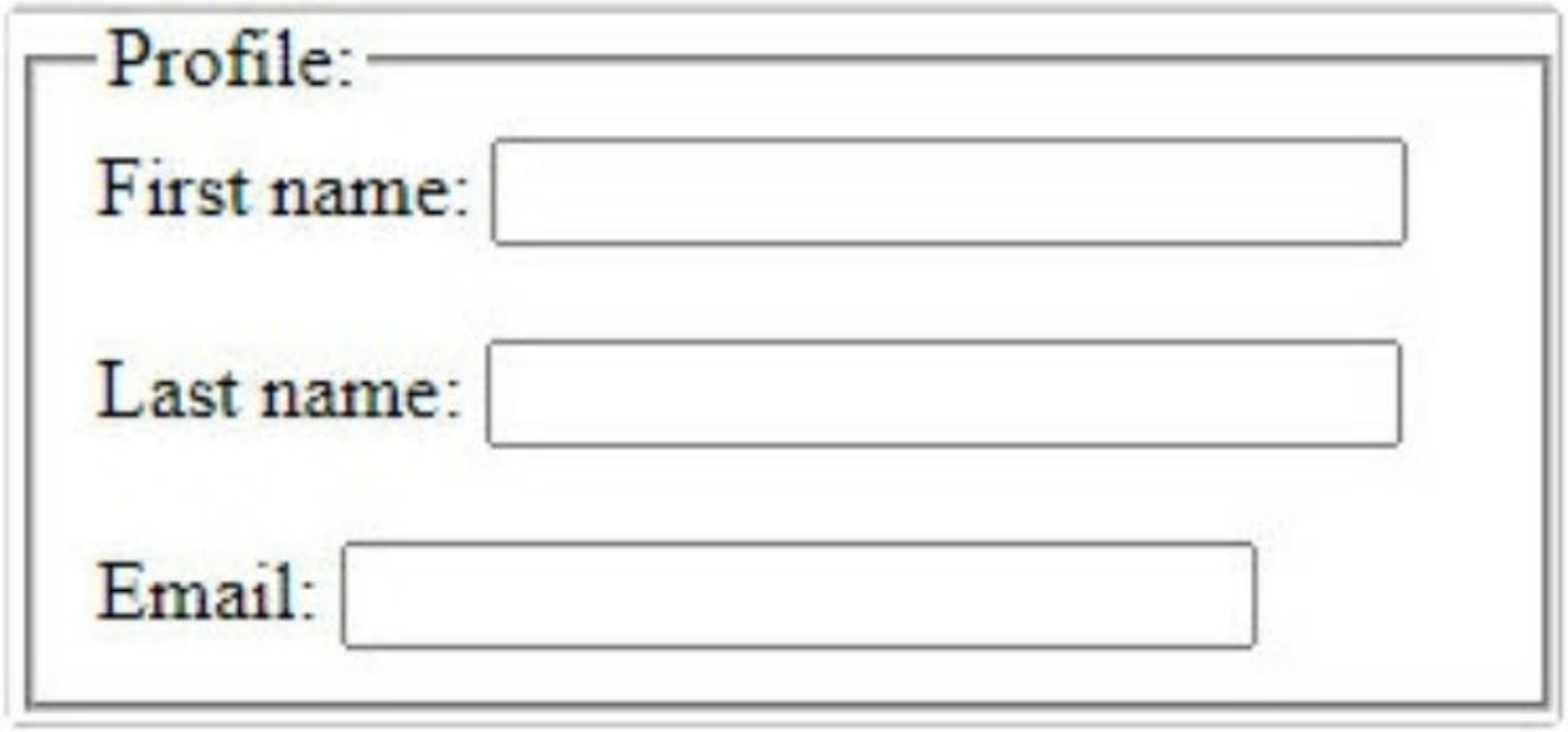
```
<button onclick="alert('Hi')">Submit</button>
```

6.1.5 <fieldset> and <legend>

The <fieldset> tag is utilized to group related elements in a form for better organization. In addition, it can contain a <legend> tag defining a caption for the elements. For example:

```
<fieldset>
  <legend>Profile:</legend>
  <label for="fname">First name:</label>
  <input type="text" id="fname" name="fname" />
  <br /><br />
  <label for="lname">Last name:</label>
  <input type="text" id="lname" name="lname" />
  <br /><br />
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" />
</fieldset>
```

Result:



6.1.6 <datalist>

The <datalist> element specifies a list of pre-defined options for an <input> element.

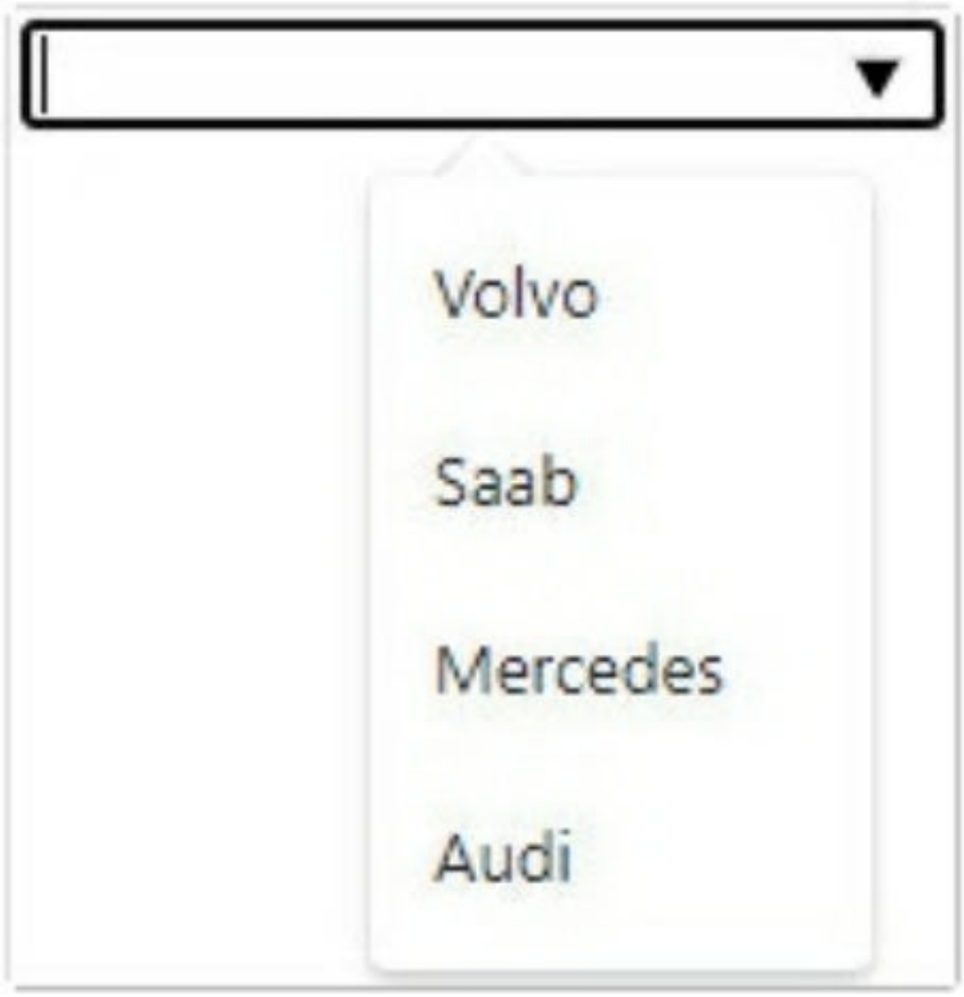
Example:

The <datalist> tag is used to specify a list of pre-defined options for an <input> element. Here is an example:

```
<datalist id="car-list">
  <option value="Volvo"></option>
  <option value="Saab"></option>
  <option value="Mercedes"></option>
  <option value="Audi"></option>
```

```
</datalist>
<input type="text" name="car" list="car-list" />
```

The code creates a text input field with the pre-defined options displayed in a drop-down list. The <datalist> tag specifies the list of options using <option> tags, and the <input> tag refers to the list using the " list " attribute, which contains the ID of the <datalist> tag.



6.2 HTML Input Types

6.2.1 Buttons

6.2.1.1 button

The <input type="button"> element creates a clickable button with the specified text, which is set using the " value " attribute. To determine which JavaScript code will run when the button is clicked, you need to utilize the " onclick " attribute. If the " onclick " attribute is not included, clicking the button will have no effect.

Here is an example:

```
<input type="button" value="Click me" onclick="alert('Hi')" />
```

This creates a button with the text "Click me"; when the button is clicked, an alert box will appear with "Hi".

6.2.1.2 submit

The <input type="submit"> element creates a submit button that submits all form values to the server when clicked. The button's text can be specified using the " value " attribute, but it is optional, with the de-

fault value being "Submit".

Here is an example:

```
<input type="submit" />
```

This creates a submit button with the default text "Submit".

6.2.1.3 reset

The <input type="reset"> element creates a reset button that resets all form values to their initial values when clicked. The button's text can be specified using the " value " attribute, but it is optional, with the default value being "Reset".

Here is an example:

```
<input type="reset" />
```

This creates a reset button with the default text "Reset".

6.2.1.4 image

The <input type="image"> element can be used as a submit button with an image. When the user

clicks on the image, the form is submitted to the server with the coordinates of where the user clicked on the image included in the request. For example, the URL for the request might look like this:

*http://127.0.0.1:5500/login?firstname=John&**x=36**&**y=25***

To use an image as a submit button, you can use the <input type="image"> element and specify the image source with the " src " attribute. The " width " and " height " attributes can be used to specify the dimensions of the image. Here's an example using a custom image called "*btn-submit.jpg*":

```
<input type="image" src="btn-submit.jpg" alt="Submit" width="50" height="50" />
```

6.2.2 Texts

6.2.2.1 text

The <input type="text"> defines a single-line text field.

Example:

```
<input type="text" name="firstname" />
```

If we type "John" into the input field, the text "John" will appear inside the field.

6.2.2.2 password

The <input type="password"> defines a password field.

Example:

```
<input type="password" name="password" />
```

When we type " 12345678 ", into the input field, we will see this:

The value of the input remains " 12345678 ", but it is not displayed on the screen because this input is a password field.

6.2.2.3 email

The <input type="email"> element creates a field for users to enter their email addresses.

Example usage:

```
<input type="email" name="email" />
```

Compared to a regular text input field, the email input field has built-in validation for email addresses. If a user enters their email address in a form, the browser will verify whether the input value is a valid email address or not. If it's not, the browser will prompt the user to enter a valid email address before submitting the form. For example:

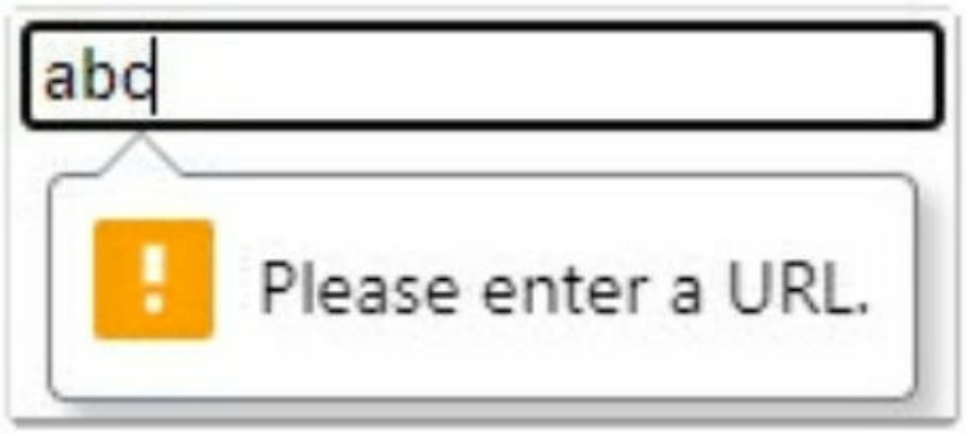
6.2.2.4 url

The <input type="url"> defines a field for entering a URL.

Example:

```
<input type="url" name="yourSite" />
```

This input type differs from regular text input as it provides basic URL validation. When a user clicks the submit button, the browser will validate the user's input to ensure that it is a valid URL. For instance:



6.2.2.5 hidden

The <input type="hidden"> element creates a hidden input field not displayed on the webpage. However, when the user clicks the submit button, the value of this input field will be forwarded to the web server along with other form data. For instance:

```
<input type="hidden" name="userId" value="123" />
```

6.2.3 Numbers

6.2.3.1 number

The <input type="number"> element creates a field where the user can only enter numeric values. In addition, the optional " min " and " max " attributes can be used to set the input's minimum and maximum allowed values. It helps to validate user input before submitting it to the server.

For example, the following code creates a number input field where the user can only enter values between 1 and 9:

```
<input type="number" name="quantity" min="1" max="9" />
```

Result:

6.2.3.2 range

The <input type="range"> element creates a slider control that allows users to select a number within a specified range. The " min " and " max " attributes are used to define the minimum and maximum values of the range. Here is an example:

```
<input type="range" name="points" min="0" max="10" />
```

Result:

6.2.4 Options

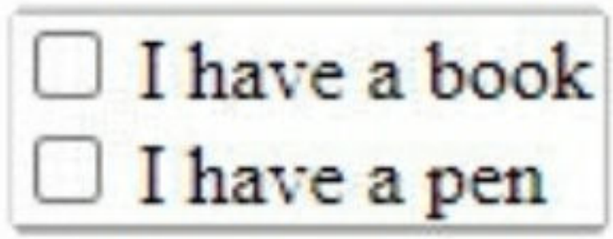
6.2.4.1 checkbox

The <input type="checkbox"> elements support selecting one or more options.

Example:

```
<input type="checkbox" id="book" name="book" value="Book" />
<label for="book"> I have a book</label>
<br />
<input type="checkbox" id="pen" name="pen" value="Pen" />
<label for="pen"> I have a pen</label>
```

Result:



We can choose none, one, or both options in the above example.

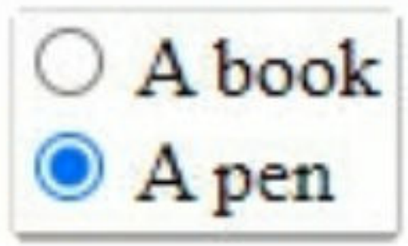
6.2.4.2 radio

The <input type="radio"> buttons can present several related options, and only one button in a group can be selected at a time. For example, consider the following code:

```
<input type="radio" id="book" name="schoolThing" value="Book" />
<label for="book"> A book</label>
<br />
```

```
<input type="radio" id="pen" name="schoolThing" value="Pen" />
<label for="pen"> A pen</label>
```

In the above example, we have two radio buttons, one for selecting a book and the other for choosing a pen.



To define these radio buttons as a group, we must give them the same " name " attribute value. In this case, both radio buttons have the " name " attribute set to " schoolThing ", which makes them part of the same group. It ensures that only one radio button can be selected within this group.

When a user selects one radio button, the previously selected button will be deselected automatically. The HTML specification for radio buttons defines this behavior.

6.2.5 Files

The <input type="file"> element enables the user to choose a file from their device for uploading to the web server. It typically includes a "**Choose File**" button to open a file selection dialog box.

Example:

```
<input type="file" name="bgimage" />
```

Result:

6.2.6 Date and Time

6.2.6.1 date

The <input type="date"> element creates a date picker field that allows users to select a date from a calendar. For instance, this is how we use it in an example:

```
<input type="date" name="birthday" />
```

The input field would then display a calendar where users can select a date by clicking on it.

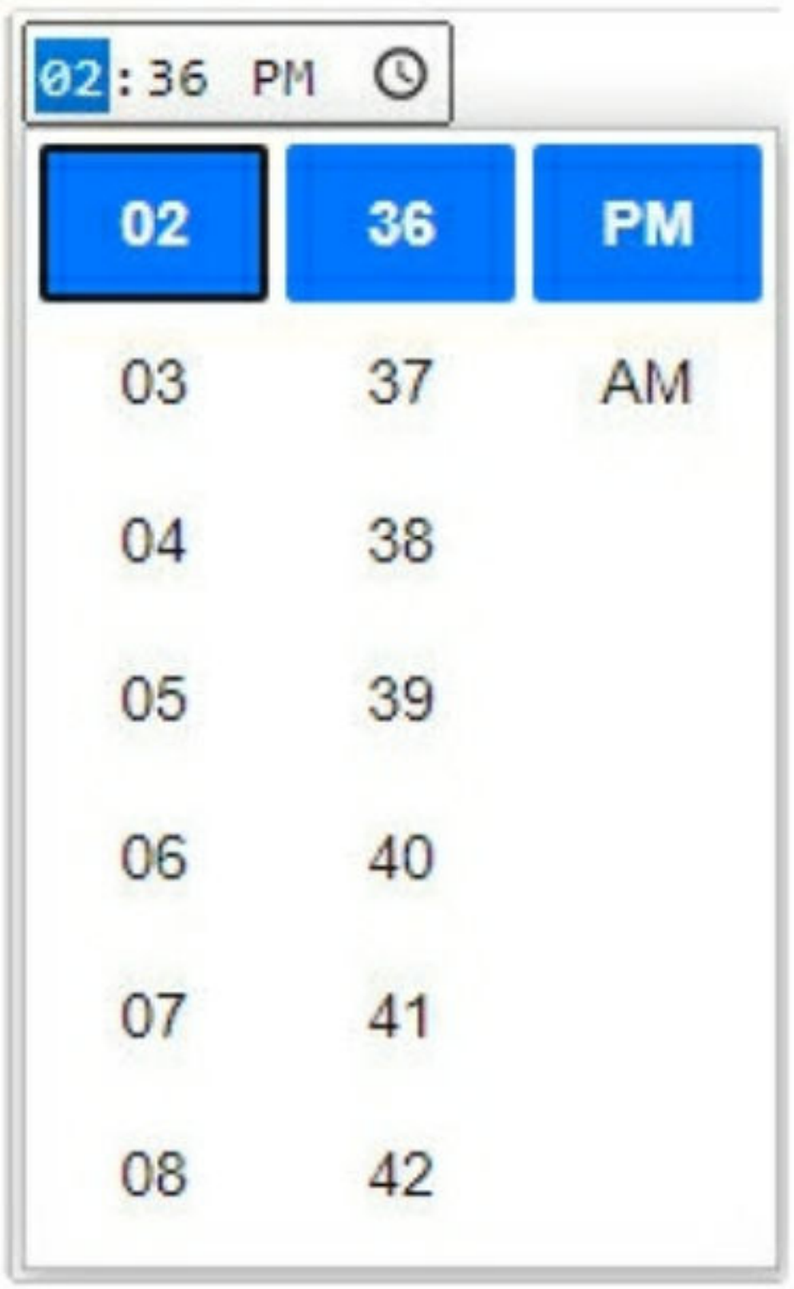

mm / dd / yyyy
September 2022
Su Mo Tu We Th Fr Sa
28 29 30 31 1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 1
2 3 4 5 6 7 8
Clear
Today

6.2.6.2 time

The <input type="time"> element provides a time picker to select the desired time. For example:

```
<input type="time" name="alarmTime" />
```

Result:



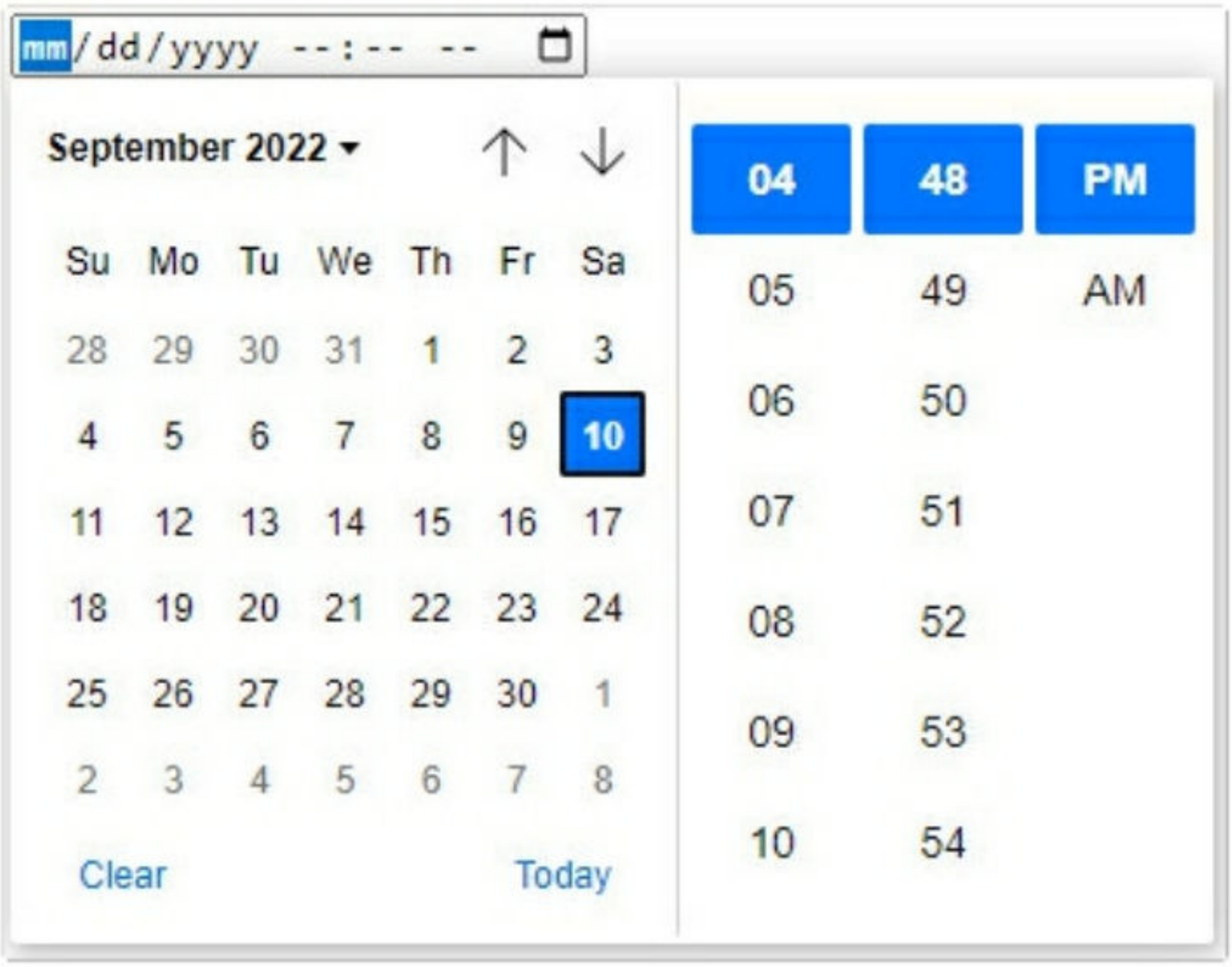
6.2.6.3 datetime-local

The <input type="datetime-local"> element allows users to select a date and time for a task or event.

Example:

```
<input type="datetime-local" name="alarmDateTime" />
```

Result:



6.2.7 Colors

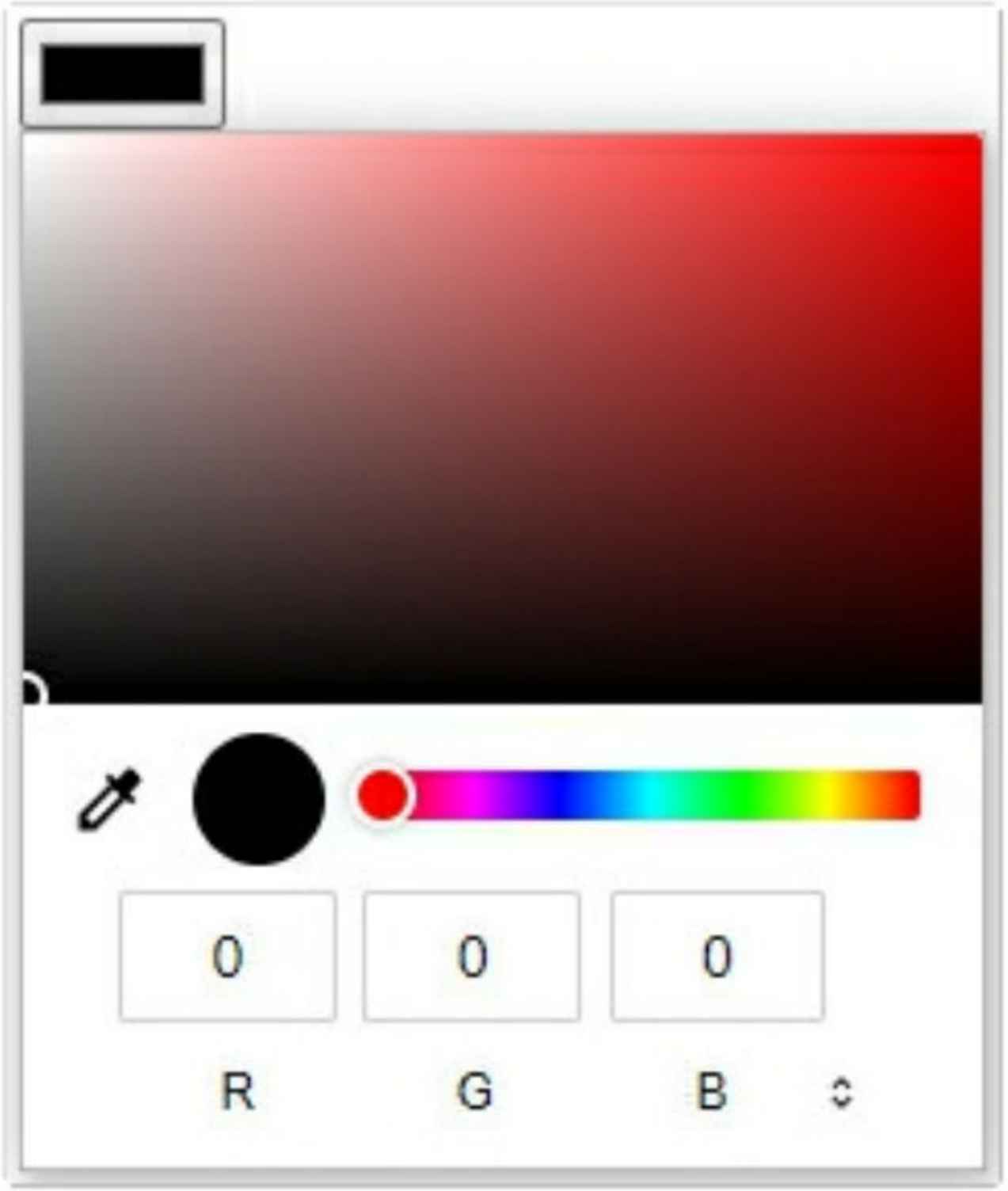
The <input type="color"> element allows users to select a color from a color palette. When the user clicks on the input field, a color picker is displayed, and the user can choose a color by clicking on it or entering a color code manually. The selected color is then assigned to the input field's value as a hexadecimal color code (e.g. #FF0000 for red).

This input type is commonly used in web development to select a background or text color for a website or application.

Example:

```
<input type="color" name="bgcolor" />
```

Result:


0
0
0
R
G
B

6.3 HTML Input Attributes

6.3.1 name

When submitting an HTML form, the " name " attribute is crucial as it acts as a parameter in the query string sent to the web server. The functioning of this attribute was previously explained at the start of this chapter.

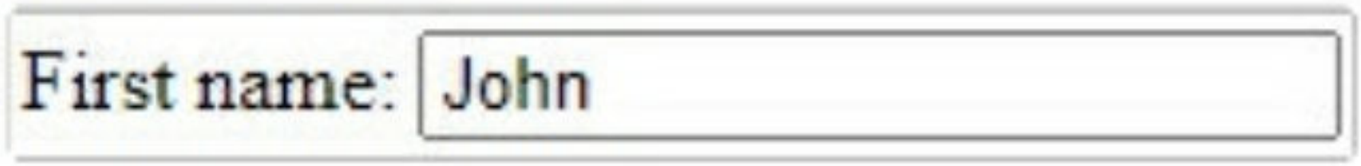
6.3.2 value

An input element's " value " attribute establishes the initial value of the element.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <form action="/login">
      <label for="fname">First name:</label>
      <input type="text" id="fname" name="firstname" value="John" />
    </form>
  </body>
</html>
```

Result:



6.3.3 placeholder

The " placeholder " attribute is utilized to indicate the anticipated value for an input element briefly. As the user inputs data, this hint is replaced with their input.

This attribute is compatible with <input> (types: text, password, email , and url) as well as <textarea> . Here is an example:

```
<input type="text" name="firstname" placeholder="Your first name here" />
```

Result:

6.3.4 readonly

The " readonly " attribute is employed to render an element unmodifiable.

Despite the un-editable nature of the element, its value will be sent to the server upon the user clicking the submit button.

Example:

```
<input type="text" name="firstname" value="John" readonly />
```

6.3.5 disabled

The " disabled " attribute renders an element unresponsive and non-clickable. This attribute can be assigned to the following elements: <input>, <button>, <textarea>, <select>, <option>, <optgroup> , and <fieldset> .

The values of disabled elements will not be transmitted to the server upon the user clicking the submit button.

Example:

```
<input type="text" name="firstname" value="John" disabled />

<input type="submit" disabled />
```

Result:

6.3.6 size

The " size " attribute specifies the character width of an <input> element. When dealing with <select> elements, the " size " attribute determines the number of visible options in the drop-down list.

Example:

```
<input type="text" name="firstname" size="6" />

<select name="car" size="2">
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

Result:

6.3.7 multiple

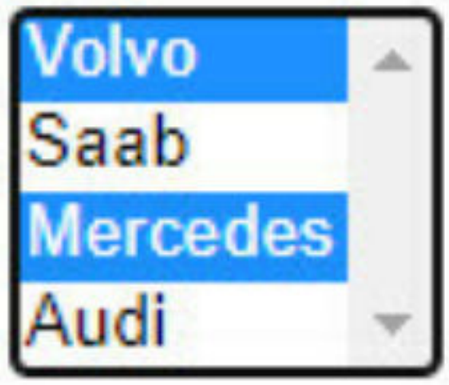
The " multiple " attribute indicates that users can input multiple values in <input> (input types: email

and file) or <select> elements.

Example:

```
<select name="cars" multiple>
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

Result:



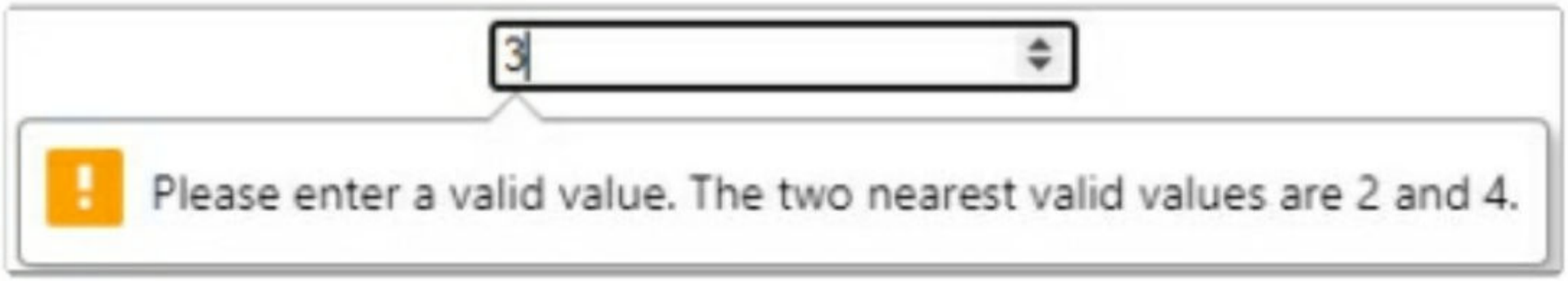
6.3.8 step

The " step " attribute defines the gap between permissible values in an <input> element. For instance, if the step is set to 2, acceptable numbers would be -2, 0, 2, 4, 6, and so on. This attribute is generally utilized with the " number " input type.

Example:

```
<input type="number" name="points" step="2" />
```

Result:



6.3.9 width and height

The " width " and " height " attributes determine the dimensions (in pixels) of the following elements: <img>, <input type="image">, <video>, <canvas> , and <iframe> .

Example:

```
<iframe src="https://www.amazon.com/dp/B09VFLS7TF" width="800" height="600"></iframe>
```

6.3.10 autofocus

The " autofocus " attribute specifies that an element should receive focus as soon as the page loads. This attribute can be assigned to the following elements: <input>, <textarea>, <select> , and <button> .

Example:

```
<input type="text" name="fname" autofocus />
```

6.3.11 autocomplete

The “ autocomplete ” attribute allows the browsers to:

- forecast the input value as users begin typing in an input field
- present suggestions to complete the field based on prior user input

This attribute collaborates with <form> and the following input types: text, password, email , and url .

Example:

```
<input type="text" name="fname" autocomplete />
```

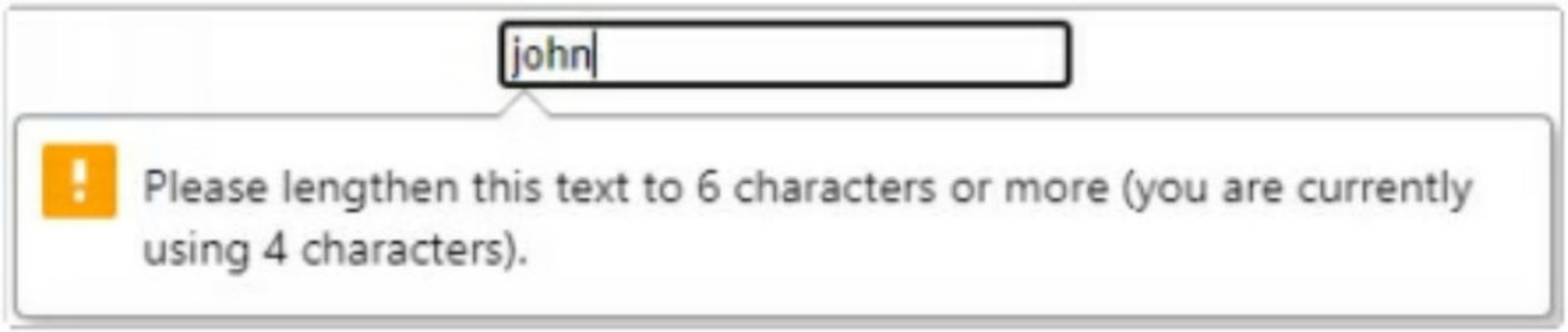
6.4 Form Validation

6.4.1 minlength

The " minlength " attribute specifies the minimum number of characters needed in an input field. This attribute is compatible with <input> (input types: text, password, email , and url) as well as <textarea> . For instance:

```
<input type="text" name="username" minlength="6" />
```

Result:



6.4.2 maxlength

The " maxlength " attribute specifies the number of characters that can be entered into an input field. This attribute works with <input> (input types: text, password, email , and url) as well as <textarea> . For example:

```
<input type="text" name="username" maxlength="6" />
```

In the example above, users are unable to input the 7th character into the field.

6.4.3 min and max

The " min " attribute indicates the minimum allowable value for <input> and <meter> elements. Conversely, the " max " attribute denotes the highest possible value for these elements.

These attributes are compatible with the input types: number, range, date, time , and datetime-local . For example:

```
<input type="number" name="quantity" min="1" max="8" />
```

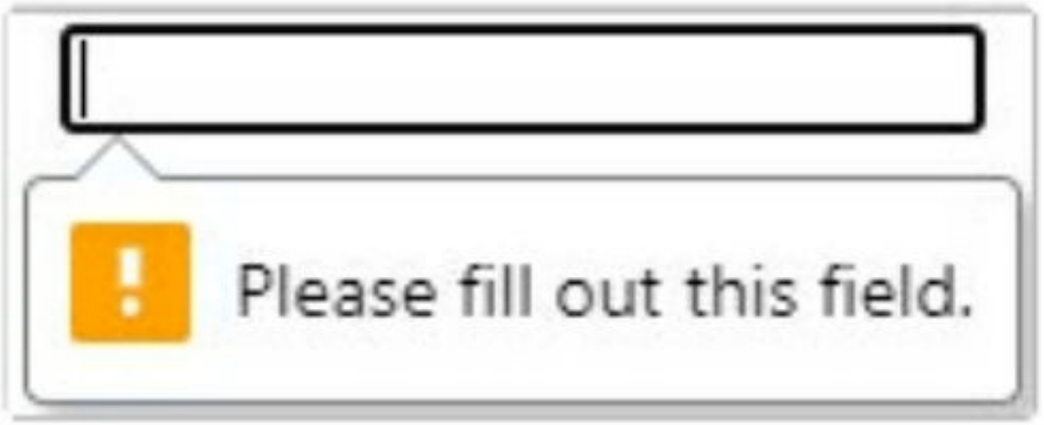
6.4.4 required

The " required " attribute demands that an input field contains a value before the form can be submitted. This attribute works with the following elements: <input>, <textarea> , and <select> .

Example:

```
<input type="text" name="fname" required />
```

Result:



6.4.5 pattern

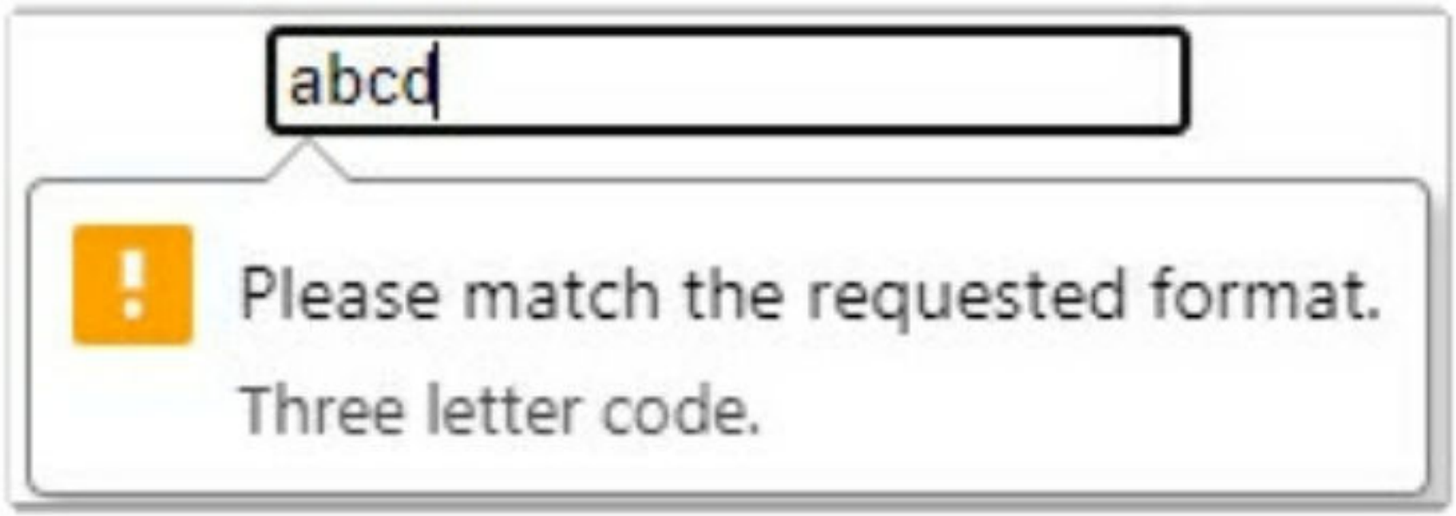
When filling out and submitting a form, the " pattern " attribute checks if the value entered in that field matches a specific pattern or regular expression. This attribute is compatible with the following input types: text, password, email , and url .

When the validation fails, the " title " attribute of the input will be used as a hint.

Example:

```
<input type="text" name="code" pattern="[A-Za-z]{3}" title="Three letter code." />
```

Result:



6.4.6 Styles for The Invalid Inputs

Certain CSS pseudo-classes enable us to emphasize the invalid inputs before submitting the form.

- " :invalid " is used to select form elements if their values are not legal according to the element's settings.
- " :required " is used to select required form elements.
- " :out-of-range " is used to select form elements with a value outside a specified range according to the min and max attributes.

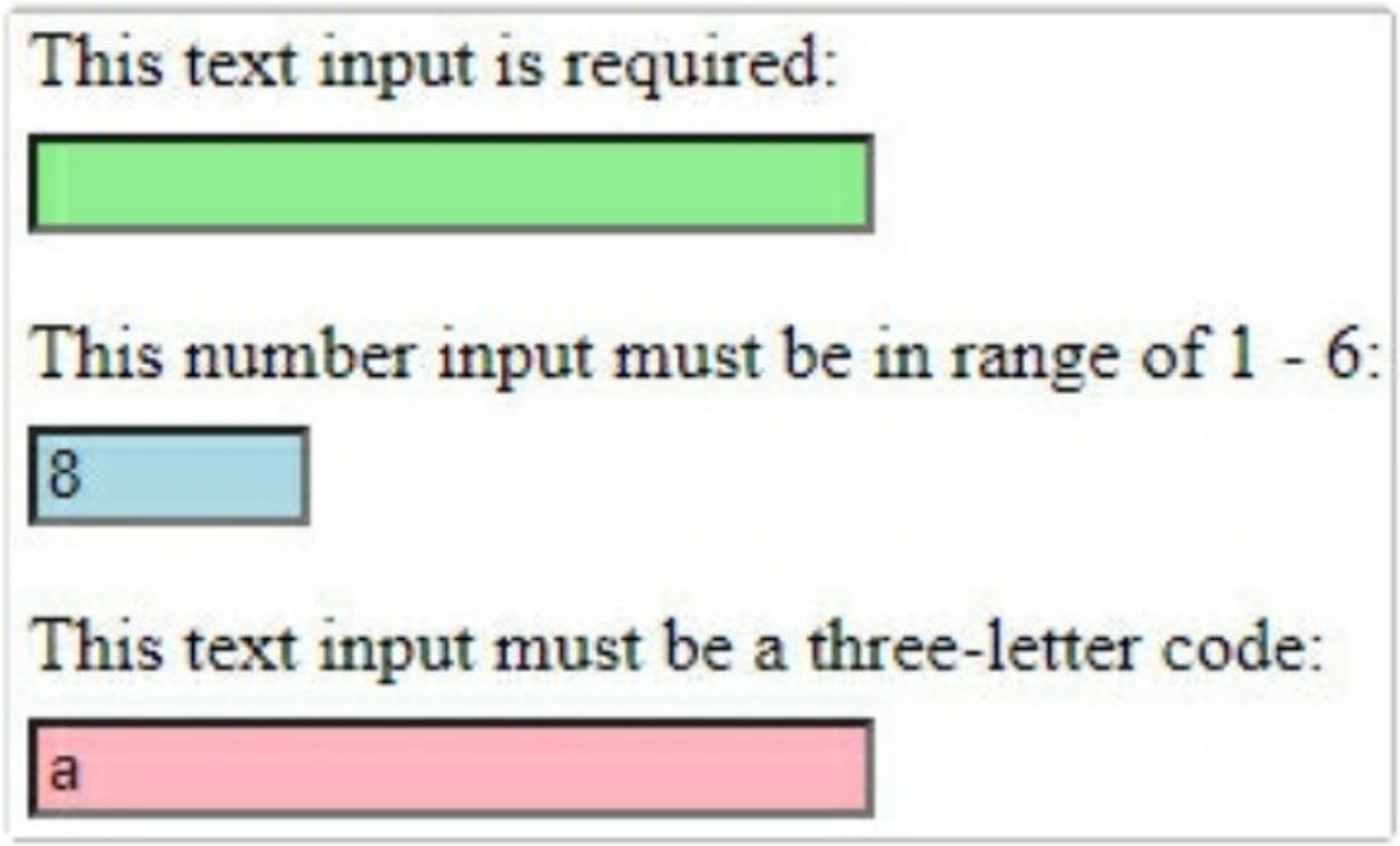
Example:

```
<!DOCTYPE html>
<html>
```

```
<head>
  <style>
    p {
      margin-bottom: 6px;
    }
    input:invalid {
      background-color: lightpink;
    }
    input:invalid:required {
      background-color: lightgreen;
    }
    input:out-of-range {
      background-color: lightblue;
    }
  </style>
</head>
<body>
  <form action="">
    <p>This text input is required:</p>
    <input type="text" name="fname" required />

    <p>This number input must be in the range of 1 - 6:</p>
```

```
        <input type="number" name="quantity" min="1" max="6" />

        <p>This text input must be a three-letter code:</p>
        <input type="text" name="code" pattern="[A-Za-z]{3}" title="Three letter code." />
      </form>
    </body>
</html>
```

Result:



CHAPTER 7

HTML Multimedia

HTML Multimedia uses multimedia elements such as images, audio, and video in web development. These elements can enhance the user experience and make the content more engaging and interactive. HTML provides a range of multimedia tags and attributes that allow developers to embed and control multimedia elements on web pages. With the increasing demand for rich media content on the web, HTML multimedia has become an essential skill for web developers.

7.1 Images

In HTML, we use the <img> element to display images. This element does not require a closing tag since it has no content. Therefore, we add the slash at the very end of the element like this:

```
<img />
```

To specify which image to be displayed, we use the " src " (which stands for "source") attribute to describe the path of the image file. For example:

```
<img src="an-apple.jpg" />
```

Another essential attribute for the <img> element is " alt " which stands for "alternative text". This attribute describes what the image is about and is crucial for search engines such as Google to know what content the image is and allow screen readers to read it out loud.

```
<img src="an-apple.jpg" alt="This is an apple" />
```

The alternative text is also used when the " src " attribute is invalid. If the path is invalid, the browser cannot download the image, and the image description from the " alt " attribute will be displayed instead. For example:

```
<img src="invalid-image.pg" alt="This is an apple" />
```

Two more attributes that are usually used in the <img> element are " width " and " height ". However, using both can distort the image because our width and height may be in a different ratio than the original image size. Therefore, using only one to scale the image up or down is best. For example, let's examine a square image of 300x300 pixels.

- This code will scale down the image:

```
<img src="my-image.jpg" alt="This is an apple" width="100" />
```

or

```
<img src="my-image.jpg" alt="This is an apple" height="100" />
```

- On the other hand, this code will scale up the image:

```
<img src="my-image.jpg" alt="This is an apple" width="500" />
```

or

```
<img src="my-image.jpg" alt="This is an apple" height="500" />
```

- But this will distort the image:

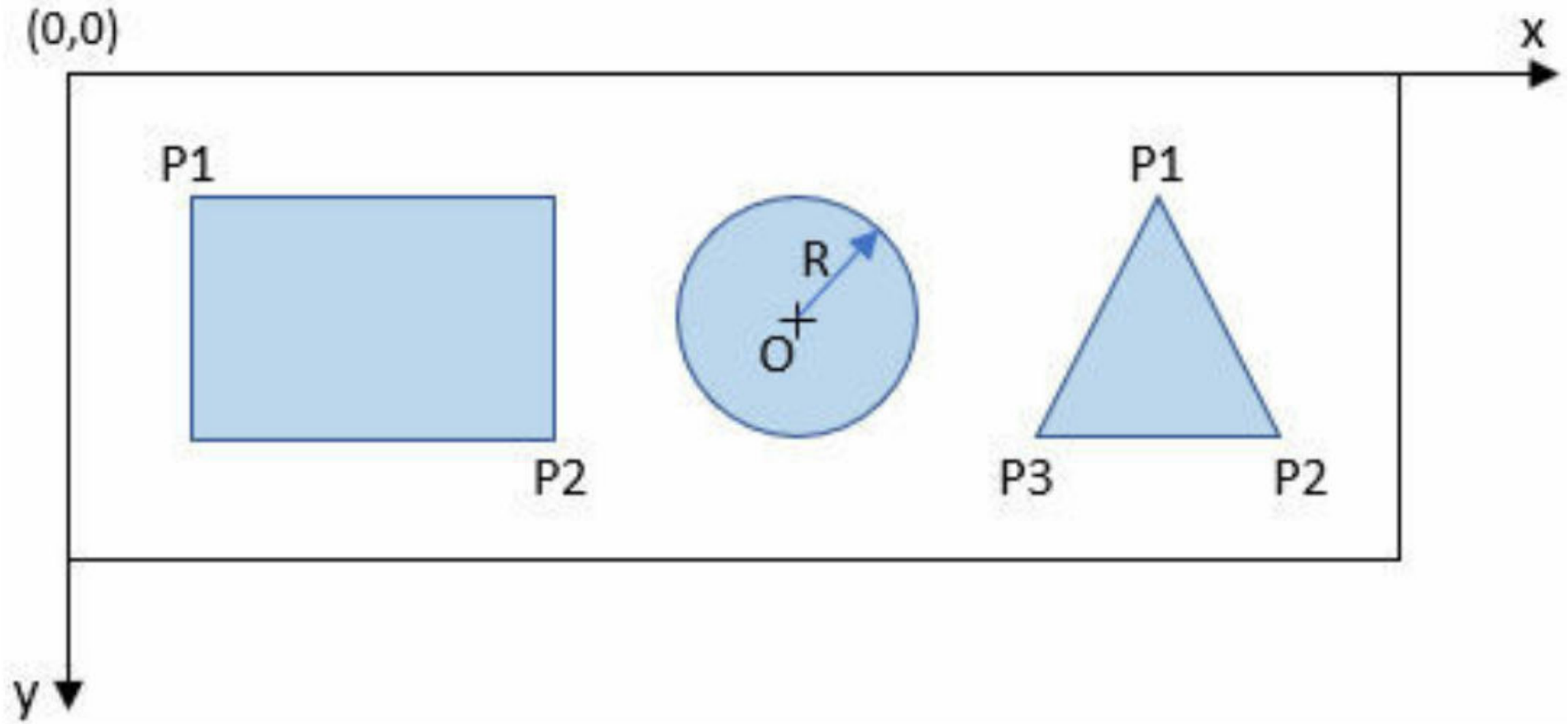
```
<img src="my-image.jpg" alt="This is an apple" width="200" height="100" />
```

7.1.1 Image Maps

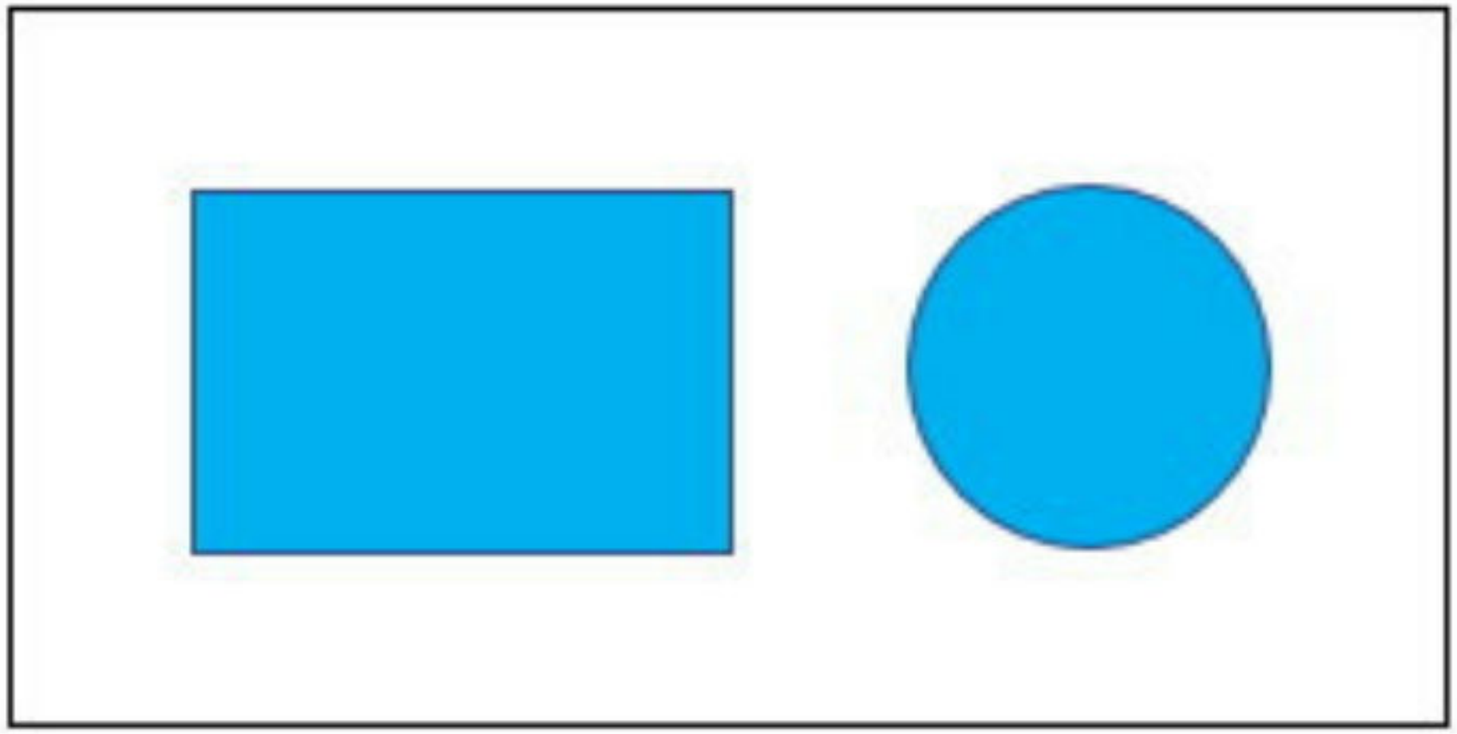
An image map is created using the <map> element, which allows specific areas of an image to be clickable. To define the clickable areas, we use one or more <area> elements as children of the <map> element. Each <map> element must have a unique " name " attribute, which is used by the <img> element's " usemap " attribute to specify which map to use.

To define a clickable area, an <area> element is used with " shape " and " coords " attributes. The " shape " attribute can take on one of the following values:

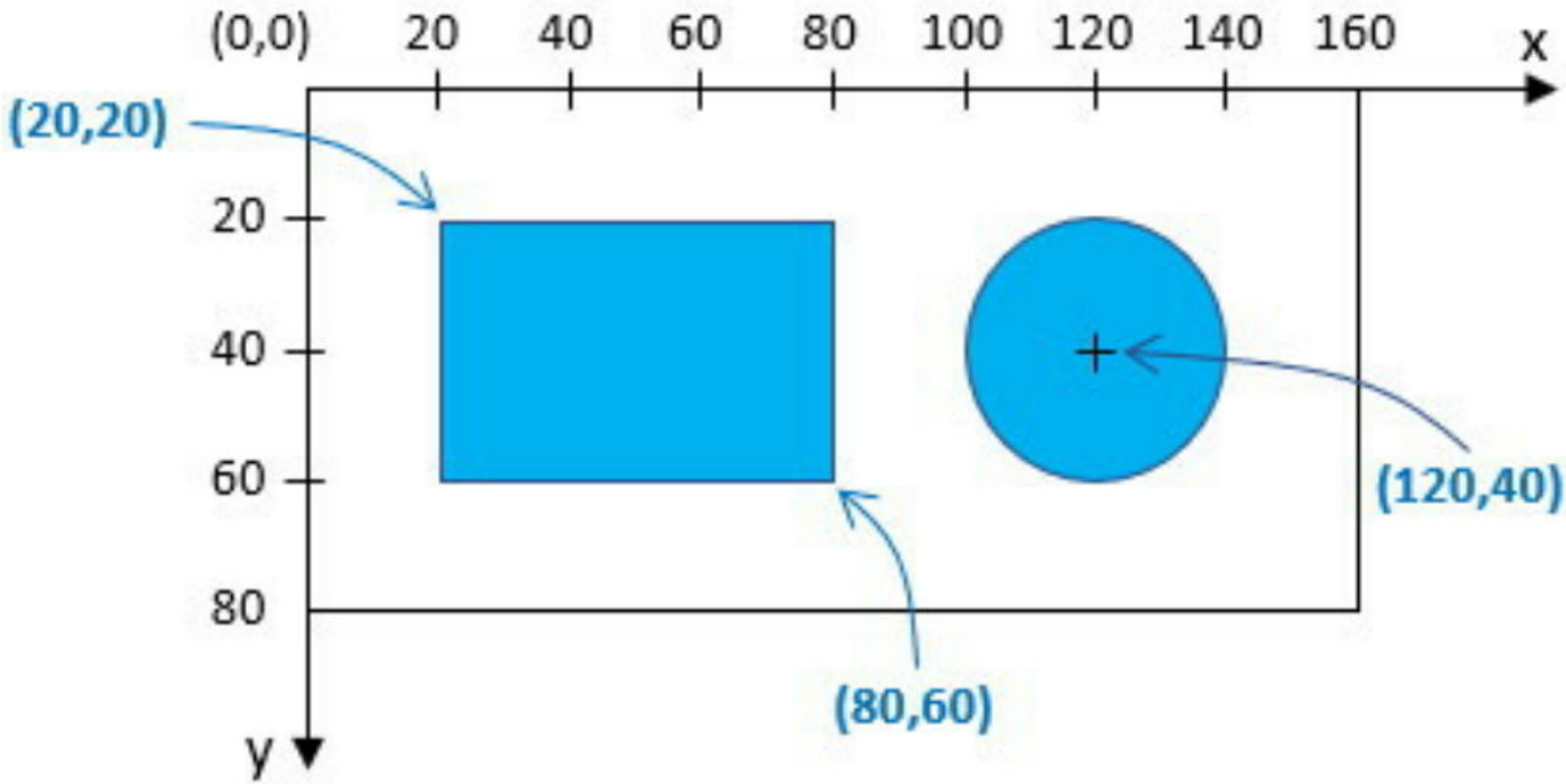
- circle - a circle. The “ cords ” attribute will be “Ox, Oy, R”.
- rect - a rectangle. The “ cords ” attribute will be “P1x, P1y, P2x, P2y”.
- poly - a polygon. The “ cords ” attribute will be “P1x, P1y, P2x, P2y$_{,\ldots,}$ PNx, PNy”.



An image map can be tested using this example image:



and the image's coordinates are as below.



index.html

```
<!DOCTYPE html>
<html>
  <body>
    <img src="navbar.jpg" alt="Navigation bar" usemap="#nav-bar" width="160" height="80" />

    <map name="nav-bar">
      <area shape="rect" coords="20,20,80,60" alt="Computer" href="#cake" />
```

```
      <area shape="circle" coords="120,40,20" alt="Coffee" href="#apple" />
    </map>

    <h2 id="apple">My Apple</h2>
    <img src="an-apple.jpg" alt="This is an apple" height="1000" />

    <h2 id="cake">My Cake</h2>
    <img src="a-cake.jpg" alt="This is a cake" height="1000" />
  </body>
</html>
```

This code is an HTML document that displays an image map with clickable areas and two images of an apple and a cake.

The first <img> tag displays an image of a navigation bar, with the "usemap" attribute specifying the name of the image map. The image is also given an " alt " attribute for accessibility purposes.

The <map> element defines the image map with a " name " attribute matching the " usemap " attribute in the <img> tag. Two <area> elements define clickable areas within the navigation bar image. When coding clickable areas, the " shape " attribute determines whether the area is a rectangle or a circle. The " coords " attribute specifies the location of the clickable area. Additionally, the " alt " attribute provides alternate text for the area, and the " href " attribute specifies the URL the area will take the user to when clicked.

After the image map, there are two <h2> tags with unique " id " attributes (" apple " and " cake "). These anchor points can be navigated by clicking the corresponding clickable areas in the image map. Finally, two <img> tags display images of an apple and a cake, respectively, with " height " attributes to scale the images up.

7.1.2 Responsive Pictures

Multiple images can be included in the <picture> element, each with its own media query, and the browser will choose one based on the query. Furthermore, if none of the <source> tags matches, the <img> element can be a backup option.

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>Resize the browser to load different images.</p>
    <picture>
      <source media="(min-width:500px)" srcset="3.jpg" />
      <source media="(min-width:300px)" srcset="2.jpg" />
      <img src="1.jpg" alt="A number" />
    </picture>
  </body>
</html>
```

The above HTML code creates a webpage containing a single paragraph and a <picture> element that displays different images based on the browser window size.

The <picture> element contains three child elements: two <source> elements and an <img> element. The <source> elements specify different image files using the " srcset " attribute and media queries with the " media " attribute. The " srcset " attribute provides the URL of an image file, while the " media " attribute specifies the minimum viewport width for each image.

The first <source> element specifies an image file named "*3.jpg*" for viewports wider than or equal to 500 pixels. The second <source> element specifies an image file named "*2.jpg*" for viewports wider than or equal to 300 pixels. If neither of these conditions is met, the <img> element will be used to display an image file named "*1.jpg*".

This approach helps optimize website image loading by serving images optimized for different screen sizes and resolutions.

7.2 Audio

The <audio> element embeds audio files into web pages. Supported audio formats include MP3, WAV, and OGG, but MP3 is preferred since most browsers support it.

Attributes of the <audio> element:

- controls - shows the control panel (play/pause, timeline, and volume)
- autoplay - plays the audio right after the page loaded

- muted – no sound mode
- loop - auto play the audio again when playing done
- preload - the audio file should be loaded when the page loads. The preload attribute is ignored if autoplay is present
- src - specifies the location (URL) of the audio file

Instead of relying solely on the " src " attribute, the <audio> element can be accompanied by one or more <source> elements to provide alternative audio files in different formats. It allows the browser to choose the most suitable audio source it can play.

Example 1:

```
<!DOCTYPE html>
<html>
  <body>
    <audio controls autoplay loop>
      <source src="my-music.ogg" type="audio/ogg" />
      <source src="my-music.mp3" type="audio/mpeg" />
      This browser does not support the audio element.
    </audio>
  </body>
</html>
```

Example 2:

```
<!DOCTYPE html>
<html>
  <body>
    <audio controls autoplay loop src="my-music.mp3"></audio>
  </body>
</html>
```

Result:

7.3 Video

The <video> element is an essential part of HTML multimedia that allows you to embed videos on a webpage. Several supported video formats exist, including MP4, WebM, and Ogg. However, MP4 is the recommended format, as most browsers widely support it.

Attributes of the <video> element:

- controls - shows the control panel (play/pause, timeline, and volume)
- autoplay - plays the audio right after the page loaded

- muted – no sound mode
- loop - auto play the audio again when playing done
- preload - the audio file should be loaded when the page loads. The preload attribute is ignored if autoplay is present
- src - specifies the location (URL) of the audio file.
- width - width of the video player
- height - height of the video player

Rather than using the " src " attribute, the <video> element can contain multiple <source> elements to specify different video formats. It allows the browser to select the best video source to handle and play it.

Example 1:

```
<!DOCTYPE html>
<html>
  <body>
    <video src="my-video.mp4" controls autoplay loop></video>
  </body>
</html>
```

Example 2:

```
<!DOCTYPE html>
<html>
  <body>
```

```
    <video width="320" height="240" autoplay>
      <source src="my-video.mp4" type="video/mp4" />
      <source src="my-video.ogg" type="video/ogg" />
      This browser does not support the video element.
    </video>
  </body>
</html>
```

CHAPTER 8

Scalable Vector Graphics

Scalable Vector Graphics or SVG is widely used for creating graphics on the web. SVG images can be scaled or resized without sacrificing quality and are commonly used as icons on web pages. Additionally, SVG files are pure XML, making them compressible for efficient transfer over the internet.

To define an SVG image, we use the <svg> element, and the image's dimensions are specified using the " width " and " height " attributes. The <svg> element typically contains several child elements for defining and drawing the picture. For instance, the following example features a <line> element.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <svg width="100" height="100">
      <line x1="0" y1="0" x2="100" y2="50" stroke="red" stroke-width="2" />
    </svg>
```

```
  </body>
</html>
```

8.1 SVG Rectangle

To draw a rectangle in an SVG image or <svg> tag, we use the <rect> element. Then, the position of its top-left corner is defined by the (x, y) coordinates, while its width and height are determined by the " width " and " height " attributes. Finally, the " fill " attribute specifies the color to fill the rectangle, while setting it to " none " will result in an outlined shape.

Example:

```
<svg width="100" height="100">
  <rect x="10" y="10" width="80" height="60" stroke="red" stroke-width="2" fill="yellow" />
</svg>
```

Result:

We can draw a rounded rectangle using the “ rx ,” and “ ry ” attributes like this.

```
<svg width="100" height="100">
  <rect x="10" y="10" width="80" height="60" rx="8" ry="8" stroke="red" stroke-width="2"
fill="yellow" />
</svg>
```

Result:

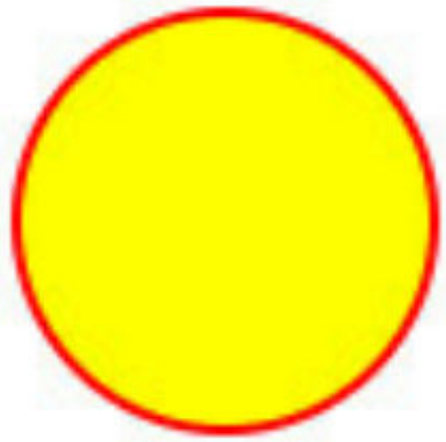
8.2 SVG Circle

To draw a circle in an SVG image or <svg> tag, we utilize the <circle> element. The circle's center is located at the coordinates (cx, cy), and its size is determined by the " r " attribute, which specifies the radius.

Example:

```
<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="red" stroke-width="2" fill="yellow" />
</svg>
```

Result:



8.3 SVG Ellipse

To create an ellipse within an SVG image or <svg> tag, we use the <ellipse> element. The center of the ellipse is located at coordinates (cx, cy), with its horizontal radius defined by the " rx " attribute and its vertical radius defined by the " ry " attribute.

Example:

```
<svg width="100" height="100">
  <ellipse cx="50" cy="50" rx="40" ry="20" fill="yellow" stroke="red" stroke-width="2" />
</svg>
```

Result:

8.4 SVG Line

To draw a line in an SVG image or <svg> tag, we employ the <line> element. The line's starting point is specified by coordinates (x1, y1), extending to a terminating point at (x2, y2). Additionally, the line won't be visible until we use the "stroke" attribute to set the stroke color.

Example:

```
<svg width="100" height="100">
  <line x1="0" y1="0" x2="100" y2="50" stroke="red" stroke-width="2" />
</svg>
```

Result:

8.5 SVG Polyline

To create a shape composed of multiple connected straight lines within an SVG image or <svg> tag, we utilize the <polyline> element. The " points " attribute of the element contains a series of (x, y) coordinates necessary to draw the polyline. For instance:

```
<svg width="100" height="100">
  <polyline points="50,0 100,50 50,100 0,50" fill="none" stroke="red" stroke-width="2" />
</svg>
```

Result:

8.6 SVG Polygon

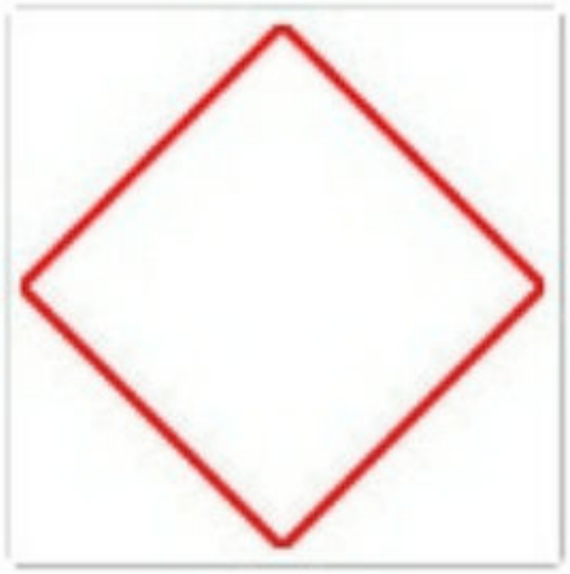
The <polygon> element is similar to a closed <polyline> as it automatically connects the last point to

the first point with a line.

Example:

```
<svg width="100" height="100">
  <polygon points="50,0 100,50 50,100 0,50" fill="white" stroke="red" stroke-width="2" />
</svg>
```

Result:



8.7 SVG Path

One of the most powerful features of SVG is the ability to create complex shapes using the <path> element. The <path> element allows for the definition of arbitrary shapes using a series of commands that control the movement of a "pen" along a virtual canvas. These commands can be used to draw lines, arcs, and curves and to define the fill and stroke properties of the resulting shape. In HTML, SVG paths can

be easily embedded and manipulated, making them a powerful tool for creating dynamic and interactive graphics on the web.

The " d " attribute, also known as path data, employs the following commands to define a path:

- M = move the pen to
- L = draw a line to
- H = draw a horizontal line to
- V = draw a vertical line to
- C = draw a curve to
- S = draw a smooth curve to
- Q = draw a quadratic Bezier curve
- T = draw a smooth quadratic Bezier curve to
- A = draw an elliptical Arc
- Z = draw a line to close the path

Uppercase letters in the " d " attribute indicate that a point is positioned at coordinates (0,0). In contrast, lowercase letters indicate that the point is positioned relative to the previous point in the path.

Example:

```
<svg height="100" width="100">
  <path d="M50 0 L25 80 l50 0 Z" fill="none" stroke="red" stroke-width="2" />
</svg>
```

Result:

Explanation:

- M50 0 - move the pen to point P1(x1, y1) = (50, 0)
- L25 80 - draw a line from the previous point (P1) to the point P2(x2, y2) = (25, 80)
- l50 0 - draw a line from the previous point (P2) to the point P3(x2 + 50, y2 + 0) = (25 + 50, 80 + 0) = (75, 80)
- Z – draw a line from the previous point (P3) to the first point (P1)

8.8 SVG Text

The <text> element defines text in an SVG image and can be customized using various attributes. For instance, we can use the " x " and " y " attributes to position the text within the image and the " fill " attribute to specify the text color. For example:

```
<svg height="100" width="200">
  <text x="0" y="20" fill="red">This is a line of text.</text>
```

```
</svg>
```

Additionally, the <text> element can contain multiple <tspan> elements, which function similarly to <text> and allow for more complex text layouts. For instance:

```
<svg height="100" width="200">
  <text x="0" y="20" fill="red">
    This is a line of text.
    <tspan x="10" y="50" fill="blue">This is the second line.</tspan>
    <tspan x="20" y="80">This is the third line.</tspan>
  </text>
</svg>
```

Transformations can also be applied to text using the " transform " attribute, which enables rotation, scaling, and other modifications. Here is an example of how to rotate text using the " transform " attribute:

```
<svg height="100" width="200">
  <text x="20" y="20" fill="red" transform="rotate(30)">This is a line of text.</text>
</svg>
```

8.9 SVG Link

A hyperlink can be created in an SVG image using the <a> element. In addition, the link's text is specified using a <text> element. Below is a code snippet example illustrating how to generate a hyperlink

in SVG:

```
<svg height="100" width="200" xmlns:xlink="http://www.w3.org/1999/xlink">
  <a xlink:href="https://www.amazon.com/dp/B09VFLS7TF" target="_blank">
    <text x="0" y="20" fill="blue">This is a link</text>
  </a>
</svg>
```

8.10 SVG Stroke

Below are stroke properties that can be used for text, lines, and shape outlines like rectangles and circles.

8.10.1 Stroke Color

The color of a stroke is defined using the " stroke " attribute.

Example:

```
<svg width="100" height="100">
  <line x1="0" y1="0" x2="100" y2="50" stroke="red" stroke-width="2" />
</svg>
```

Result:

8.10.2 Stroke Width

Similar to the above example, the thickness of a stroke is defined using the " stroke-width " attribute.

8.10.3 Stroke Ending

The type of endings for an open path can be defined using the " stroke-linecap " attribute, which can take one of the following values: " butt " for no stops (default), " square " for square stops, and " round " for round stops.

For example, in the SVG code snippet provided below, four lines are drawn with different “ stroke-linecap ” values:

```
<svg width="100" height="100">
  <line x1="10" y1="10" x2="90" y2="10" stroke="red" stroke-width="6" />
  <line x1="10" y1="30" x2="90" y2="30" stroke="red" stroke-width="6" stroke-linecap="butt" />
  <line x1="10" y1="50" x2="90" y2="50" stroke="red" stroke-width="6" stroke-linecap="square" />
  <line x1="10" y1="70" x2="90" y2="70" stroke="red" stroke-width="6" stroke-linecap="round" />
</svg>
```

Result:

The square and round endings of the third and fourth lines cause them to extend beyond their actual length of 80 pixels.

8.10.4 Dash Stroke

To create dashed lines, we use the " stroke-dasharray " attribute. Then, the line segments are drawn using the odd-indexed items in the dash array, while the even-indexed items specify the spaces between them. For example:

```
<svg width="300" height="100">
  <line x1="0" y1="10" x2="300" y2="10" stroke="red" stroke-width="2" stroke-dasharray="10,5,20" /
>
</svg>
```

Result:

Explanation:

- 10 was used to draw the 1st line segment
- 5 was used to make the 1st space
- 20 was used to draw the 2nd line segment

since the line is 300px in length, it continues to use the dash array to draw the rest

- 10 was used to make the 2nd space
- 5 was used to draw the 3rd line segment
- 20 was used to make the 3rd space

and

- 10 was used to draw the 4th line segment
- 5 was used to make the 4th space
- 20 was used to draw the 5th line segment

and so on.

8.11 SVG Gradients

8.11.1 Linear Gradient

An SVG image can be filled with a linear gradient using the <linearGradient> element. This element is enclosed within a <defs> tag, which can contain multiple definitions. The gradient comprises two or more colors, each defined with a <stop> element nested within the <linearGradient> tag.

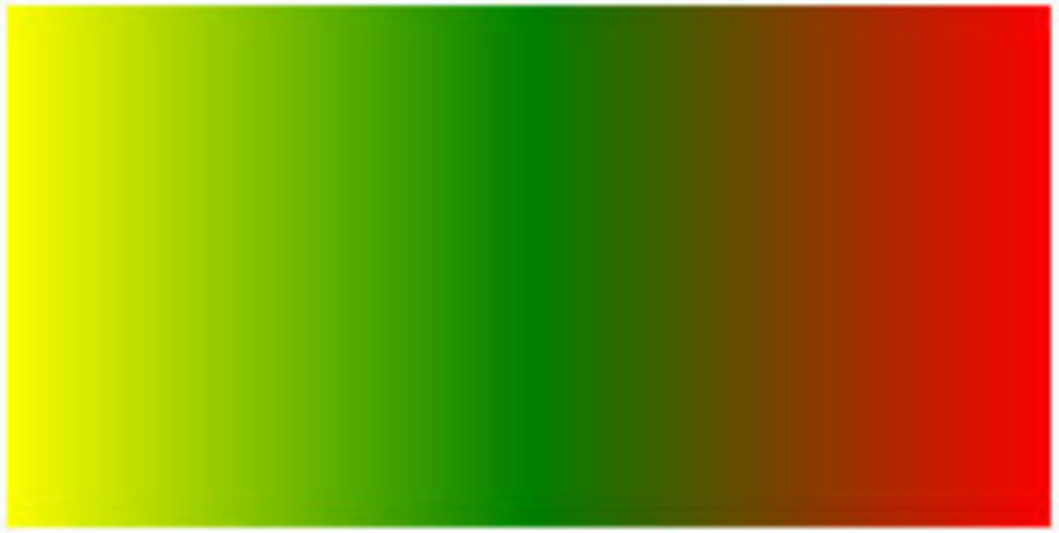
The <linearGradient> element has two attributes, (x1,y1) and (x2,y2), which define the start and end points of the gradient. Depending on the values of these attributes, the gradient can be horizontal, vertical, or diagonal:

- The gradient is horizontal if x1 and x2 are different and y1 and y2 are equal.
- The gradient is vertical if x1 and x2 have the same value, but y1 and y2 are not identical.
- The gradient is diagonal if both x1 and x2 and y1 and y2 are distinct values.

Example 1:

```
<svg height="100" width="200">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="100%" y2="0%">
      <stop offset="0%" stop-color="yellow" />
      <stop offset="50%" stop-color="green" />
      <stop offset="100%" stop-color="red" />
    </linearGradient>
  </defs>
  <rect x="0" y="0" width="200" height="100" fill="url(#grad)" />
</svg>
```

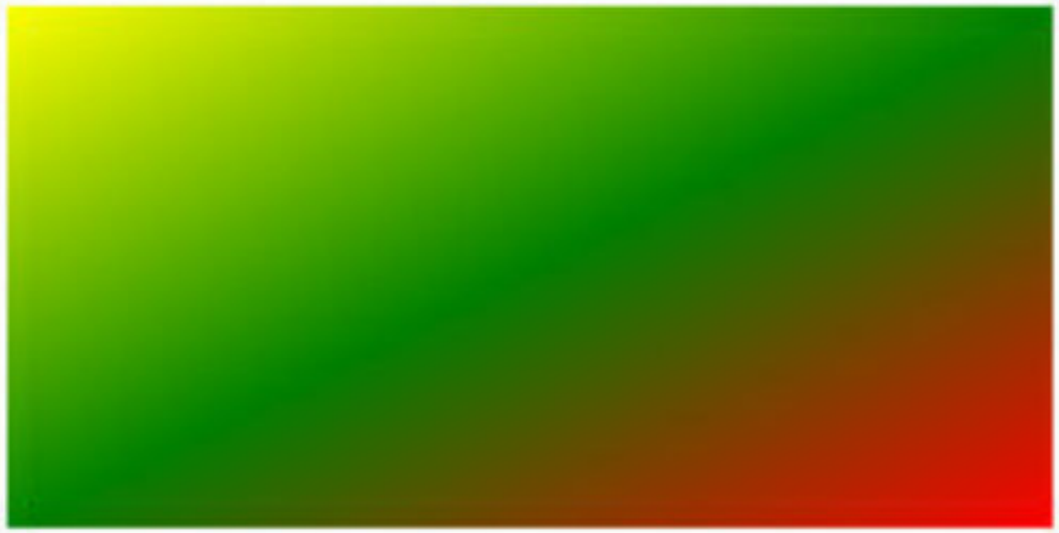
Result:



Example 2:

```
<svg height="100" width="200">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="100%" y2="100%">
      <stop offset="0%" stop-color="yellow" />
      <stop offset="50%" stop-color="green" />
      <stop offset="100%" stop-color="red" />
    </linearGradient>
  </defs>
  <rect x="0" y="0" width="200" height="100" fill="url(#grad)" />
</svg>
```

Result:



8.11.2 Radial Gradient

In SVG images, a radial gradient can fill a shape with color using the <radialGradient> element. This element is enclosed within a <defs> tag, which can contain multiple definitions. The gradient comprises two or more colors, each defined with a <stop> element nested within the <radialGradient> tag.

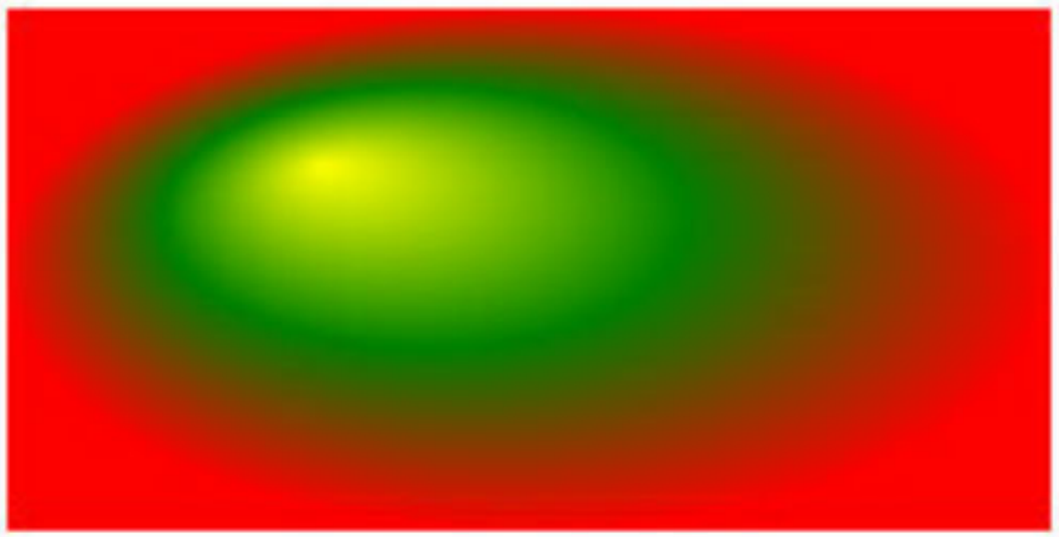
The <radialGradient> element has several attributes. The (cx,cy) attributes determine the gradient's center, while the " r " attribute specifies the radius of the outermost circle. The (fx,fy) attributes determine the center of the innermost circle.

Example:

```
<svg height="100" width="200">
  <defs>
    <radialGradient id="grad" cx="50%" cy="50%" r="50%" fx="30%" fy="30%">
      <stop offset="0%" stop-color="yellow" />
      <stop offset="50%" stop-color="green" />
```

```
        <stop offset="100%" stop-color="red" />
      </radialGradient>
    </defs>
    <rect x="0" y="0" width="200" height="100" fill="url(#grad)" />
</svg>
```

Result:



8.12 SVG Filters

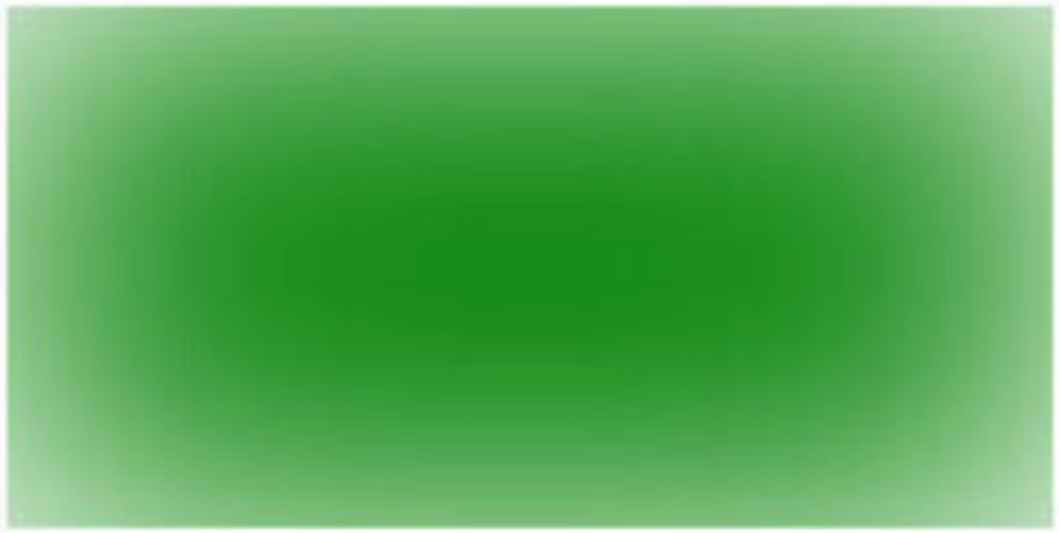
An SVG filter can be defined using the <filter> element. In the example below, various filters are available in SVG, but we will focus on the Gaussian Blur filter, implemented using the <feGaussianBlur> element.

The amount of blur can be defined using the " stdDeviation " attribute of the <feGaussianBlur> element. In contrast, the " in " attribute with a value of " SourceGraphic " specifies that the effect should be applied to the entire element.

Example:

```
<svg height="100" width="200">
  <defs>
    <filter id="my-filter">
      <feGaussianBlur in="SourceGraphic" stdDeviation="30" />
    </filter>
  </defs>
  <rect x="0" y="0" width="200" height="100" fill="green" filter="url(#my-filter)" />
</svg>
```

Result:



CHAPTER 9

HTML Canvas

HTML5 Canvas is a powerful tool that allows web developers to create dynamic and interactive graphics on their web pages. With Canvas, developers can draw and manipulate images, animations, and even videos in real-time using just a few lines of code. This technology has revolutionized how we think about web development, opening up new possibilities for creating engaging and immersive user experiences.

This section will explore the basics of HTML5 Canvas, including how to create and manipulate shapes, add text and images. Upon completing this section, you'll have a solid understanding of using Canvas to bring your web pages to life and engage your users in exciting new ways.

9.1 Drawing Lines and Paths

The <canvas> element in HTML5 is used to draw webpage graphics. To enable JavaScript to reference the canvas later, it always has an " id " attribute. Additionally, the " width " and " height " attributes define the canvas size.

By default, the canvas has a transparent background and no border. This allows it to be placed in front of other HTML elements, such as videos, and rectangles can be drawn to highlight objects, such as cars on the road.

Here is an example of how to use the `<canvas>` element:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <canvas id="my-canvas" width="800" height="600">This browser does not support the canvas</
canvas>

    <script src="./app.js"></script>
  </body>
</html>
```

This code creates a primary HTML document with a canvas element and references to an external stylesheet and a JavaScript file. When the page loads, the JavaScript code in "*app.js*" will be executed, manipulating the canvas element and drawing graphics. The text " This browser does not support the canvas " will be displayed if the browser doesn't support the canvas element.

style.css

```
#my-canvas {
    border: 1px solid #ccc;
}
```

app.js

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    console.log(context);
};
```

This JavaScript code sets an event listener for the " onload " event of the “ window ” object, which means that the code will run after all the elements on the page have loaded. In addition, it ensures the canvas element is available for the JavaScript code to access.

Afterwards, it employs the document.getElementById() function to fetch the canvas element with the ID of " my-canvas " and subsequently assign it to the “ canvas ” variable.

Then, the canvas.getContext() method is used to get a rendering context for the canvas. The argument " 2d " is passed in this case, meaning the code will get a 2D rendering context. Finally, this context is assigned to the “ context ” variable.

Finally, the code logs the context variable to the console, which is for debugging purposes. Overall, this code retrieves the canvas element and its rendering context, which is necessary for drawing graphics on the canvas.

9.1.1 Drawing Lines

Using JavaScript, the HTML canvas element is a powerful tool for creating graphics and visual effects on a webpage. Drawing lines is one of the fundamental tasks in creating graphics, and the canvas element provides several methods to do it.

The first method for drawing lines is the `moveTo()` method. This method sets the starting point of the line. For example, the following code sets the starting point to `(50,50)`:

```
context.moveTo(50, 50);
```

The `lineTo()` method is then used to draw a line from the starting point to a new point. For example, the following code draws a line from `(50,50)` to `(150,150)`:

```
context.lineTo(150, 150);
```

To render the line on the canvas, the `stroke()` method is used:

```
context.stroke();
```

The `stroke()` method applies the current stroke style to the line and draws it on the canvas.

Here's an example that puts it all together to draw a diagonal line on a canvas element:

app.js

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    context.beginPath();
    context.moveTo(50, 50);
    context.lineTo(150, 150);
    context.stroke();
};
```

9.1.1.1 Stroke Styles

The “ strokeStyle ” property is a fundamental part of the HTML canvas that allows you to set the stroke color used when drawing shapes or lines on the canvas.

The syntax for setting the “ strokeStyle ” property is straightforward:

```
context.strokeStyle = color;
```

Here, context is the canvas context object, and color is a string that specifies the stroke's color. You can set the color using a variety of values, such as a color name (" red "), a hexadecimal value (" #FF0000 "), or an RGB value (" rgb(255, 0, 0) ").

Basic Usage

Let's start with a basic example that demonstrates how to use the “ strokeStyle ” property to set the color of a stroke:

```
window.onload = function () {
   const canvas = document.getElementById("my-canvas");
   const context = canvas.getContext("2d");

   context.lineWidth = 20;

   // Set the stroke color
   context.strokeStyle = "red";

   // Draw a rectangle with a red stroke
   context.strokeRect(50, 50, 200, 200);
};
```

In this example, we set the “ strokeStyle ” property to " red " using the string " red ". We then draw a rectangle using the strokeRect() method, which draws a rectangular outline with the current stroke style.

Gradients and Patterns

In addition to setting the strokeStyle property to a solid color, you can also set it to a gradient or pattern to create more complex stroke styles.

Gradients

To create a gradient stroke, you can use the createLinearGradient() or createRadialGradient() methods of the canvas context object to create a gradient object. Then, set the " strokeStyle " property to that gradient object.

Here's an example of using the createLinearGradient() method to create a linear gradient stroke:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");

  context.lineWidth = 20;

  // Create a linear gradient stroke
  const gradient = context.createLinearGradient(50, 50, 200, 200);
  gradient.addColorStop(0, "red");
  gradient.addColorStop(1, "blue");
  context.strokeStyle = gradient;

  // Draw a rectangle with a gradient stroke
  context.strokeRect(50, 50, 200, 200);
};
```

In this example, we create a linear gradient stroke using the createLinearGradient() method, which takes four arguments representing the start and end points of the gradient. We then add two color stops to

the gradient using the addColorStop() method, which takes a position value between 0 and 1 and a color value.

Patterns

To create a patterned stroke, you can use the createPattern() method of the canvas context object to create a pattern object, then set the " strokeStyle " property to that pattern object.

Here's an example of using the createPattern() method to create a patterned stroke:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");

  context.lineWidth = 20;

  // Create a pattern stroke
  const image = new Image();
  image.src = "pattern.png";
  image.onload = () => {
    const pattern = context.createPattern(image, "repeat");
    context.strokeStyle = pattern;

    // Draw a rectangle with a pattern stroke
    context.strokeRect(50, 50, 200, 200);
  };
```

```
};
```

In this example, we create a patterned stroke using the createPattern() method, which takes two arguments: an image object and a repetition value ("repeat", "repeat-x", "repeat-y", or "no-repeat"). We then set the “ strokeStyle ” property to the pattern object.

We load an image using the Image constructor and set the onload property to a function that creates the pattern and draws the rectangle once the image has loaded. It is necessary because the createPattern() method requires a fully loaded image object.

9.1.1.2 Dashed or Dotted Lines

The setLineDash() method is a feature of the HTML canvas that allows you to create dashed or dotted lines when drawing shapes or lines on the canvas. This section will discuss using the setLineDash() method to create dashed lines and provide examples to illustrate its usage.

The setLineDash() Method

The setLineDash() method takes a single argument, an array of numbers specifying each dash's length and gap in the dashed line. The array can contain any number of elements, with even-indexed elements representing the dash length and odd-indexed elements representing the gap length. So, for example, the array [5, 10] would create a dashed line with a dash length of 5 pixels followed by a gap length of 10 pixels, and this pattern would repeat for the entire length of the line.

Here is an example of using the setLineDash() method to create a dashed line:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    // Set the line dash pattern to [5, 10]
    context.setLineDash([5, 10]);

    // Draw a dashed line
    context.beginPath();
    context.moveTo(10, 10);
    context.lineTo(100, 10);
    context.stroke();
};
```

In this example, we set the lineDash property of the context to an array of [5, 10], which specifies a dash length of 5 pixels followed by a gap length of 10 pixels. Finally, we create a path consisting of two connected points using the moveTo() and lineTo() methods and stroke the path using the stroke() method. The resulting line will be a dashed line with a pattern of 5-pixel dashes followed by 10-pixel gaps.

Applying the setLineDash() Method to Shapes

The setLineDash() method can create dashed lines for any shape drawn on the canvas, including rectangles, arcs, and curves. Here's an example of drawing a dashed rectangle using the setLineDash() method:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    // Set the line dash pattern to [5, 10]
    context.setLineDash([5, 10]);

    // Draw a dashed rectangle
    context.beginPath();
    context.rect(10, 10, 50, 50);
    context.stroke();
};
```

In this example, we create a dashed rectangle by setting the dash pattern using the setLineDash() method. Using the stroke() method to apply the line style, we then stroke the rectangle.

Resetting The Line Dash

We can also reset the line dash pattern to its default value of a solid line by calling the setLineDash() method with an empty array, like so:

```
context.setLineDash([]);
```

9.1.1.3 Line Caps

When you draw a line on an HTML canvas using JavaScript, the ends of the line can have different styles.

These styles are known as "line caps," and they determine how the ends of the line are rendered.

The HTML canvas element provides three different line caps: " butt ", " round ", and " square ". The default line cap is " butt ". Here's how to use each of these line caps:

- **Butt:** This is the default line cap. A line with a " butt " cap will end abruptly at the endpoint of the line. You can set the line cap to " butt " by setting the “ lineCap ” property to the value " butt " like this:

```
context.lineCap = "butt";
```

- **Round:** A line with a " round " cap will end with a semi-circle with a radius equal to the line width. You can set the line cap to " round " by setting the “ lineCap ” property to the value " round " like this:

```
context.lineCap = "round";
```

- **Square:** A line with a " square " cap will end with a rectangle with a length equal to the line width and a width equal to half the line width. You can set the line cap to " square " by setting the “ lineCap ” property to the value " square " like this:

```
context.lineCap = "square";
```

Here's an example that shows each of these line caps in action:

app.js

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
```

```
const context = canvas.getContext("2d");

context.lineWidth = 10;

context.beginPath();
context.moveTo(50, 50);
context.lineTo(150, 50);
context.stroke();

context.lineCap = "round";
context.beginPath();
context.moveTo(50, 100);
context.lineTo(150, 100);
context.stroke();

context.lineCap = "square";
context.beginPath();
context.moveTo(50, 150);
context.lineTo(150, 150);
context.stroke();
};
```

In this example, we set the line width to 10 pixels using the “ linewidth ” property and begin a new path using the beginPath() method. We then draw a line from (50,50) to (150,50) using the moveTo() and lineTo() methods and apply the current stroke style to the line using the stroke() method.

We then change the line cap to " round " using the “ lineCap ” property and draw a new line from (50,100) to (150,100) .

Finally, we change the line cap to " square " using the “ lineCap ” property and draw a new line from (50,150) to (150,150) .

Overall, using line caps in HTML canvas is a simple way to add different styles to the ends of lines. Using the “ lineCap ” property, you can easily change the line cap and create a wide range of visual effects on a canvas element.

9.1.1.4 Line Joins

When you draw lines with the HTML canvas element, the points where the lines meet can have different styles. These styles are known as "line joins", and they determine how the corners of a path are rendered. The HTML canvas element provides three different line joins: " miter ", " round ", and " bevel ". The default line join is " miter ". Here's how to use each of these line joins:

- **Miter:** This is the default line join. A line with a " miter " join will have a sharp corner where the lines meet. You can set the line join to " miter " by assigning the “ lineJoin ” property the value " miter " like this:

```
context.lineJoin = "miter";
```

- **Round:** A line with a " round " join will have a rounded corner where the lines meet. You can set the line join to " round " by assigning the “ lineJoin ” property the value " round " like this:

```
context.lineJoin = "round";
```

- Bevel: A line with a " bevel " join will have a flat corner where the lines meet, with a diagonal line connecting the two end points. You can set the line join to " bevel " by assigning the “ lineJoin ” property the value " bevel " like this:

```
context.lineJoin = "bevel";
```

Here's an example that shows each of these line joins in action:

app.js

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    context.lineWidth = 20;

    context.lineJoin = "miter";
    context.beginPath();
    context.moveTo(50, 20);
    context.lineTo(150, 20);
    context.lineTo(150, 60);
```

```
    context.stroke();

    context.lineJoin = "round";
    context.beginPath();
    context.moveTo(50, 80);
    context.lineTo(150, 80);
    context.lineTo(150, 120);
    context.stroke();

    context.lineJoin = "bevel";
    context.beginPath();
    context.moveTo(50, 140);
    context.lineTo(150, 140);
    context.lineTo(150, 180);
    context.stroke();
};
```

In this example, we set the line width to 20 pixels using the “ linewidth ” property and begin a new path using the beginPath() method. We then draw a line from (50,20) to (150,20) and from (150,20) to (150,60) using the moveTo() and lineTo() methods and apply the current stroke style to the line using the stroke() method.

We then change the line join to " round " using the “ lineJoin ” property and draw a new line from (50,80) to (150,80) and from (150,100) to (150,120).

Finally, we change the line join to " bevel " using the “ lineJoin ” property and draw a new line from (50,140) to (150,140) and from (150,140) to (150,180).

As you can see, each line join has a distinct visual style. The " miter " join creates a sharp corner, which can be helpful when drawing geometric shapes. The " round " join creates a smooth, curved corner, which can be helpful when drawing organic shapes or illustrations. Finally, the " bevel " join creates a flat corner with a diagonal line connecting the two endpoints, which can be helpful when drawing shapes with angles that aren't too sharp.

9.1.2 Drawing Curves

Drawing curves in HTML canvas is essential for creating various visual effects, from simple arcs to complex shapes. Several methods are available in the Canvas API for drawing curves, including arc, arcTo, bezierCurveTo , and quadraticCurveTo . Let's look at these methods and how they can be used.

9.1.2.1 The arc method

The arc(x, y, r, startAngle, endAngle, isCounterClockwise) method is used to draw circular or part-circular curves. It takes six arguments:

- the x and y coordinates of the center of the circle
- the radius of the circle
- the starting angle
- the ending angle
- a Boolean value that indicates whether the arc should be drawn in a counterclockwise direction

Here's an example that draws a half-circle:

```
window.onload = function () {
   const canvas = document.getElementById("my-canvas");
   const context = canvas.getContext("2d");

   context.beginPath();
   context.arc(100, 100, 50, 0, Math.PI, true);
   context.stroke();
};
```

In this example, we first call the `beginPath()` method to start a new path. Then, we call the `arc()` method with the center coordinates `(100,100)`, a radius of `50`, a starting angle of `0`, and an ending angle of `Math.PI` (which is equivalent to 180 degrees), and a counterclockwise direction. Finally, we call the `stroke()` method to stroke the path with the current stroke style.

9.1.2.2 The arcTo method

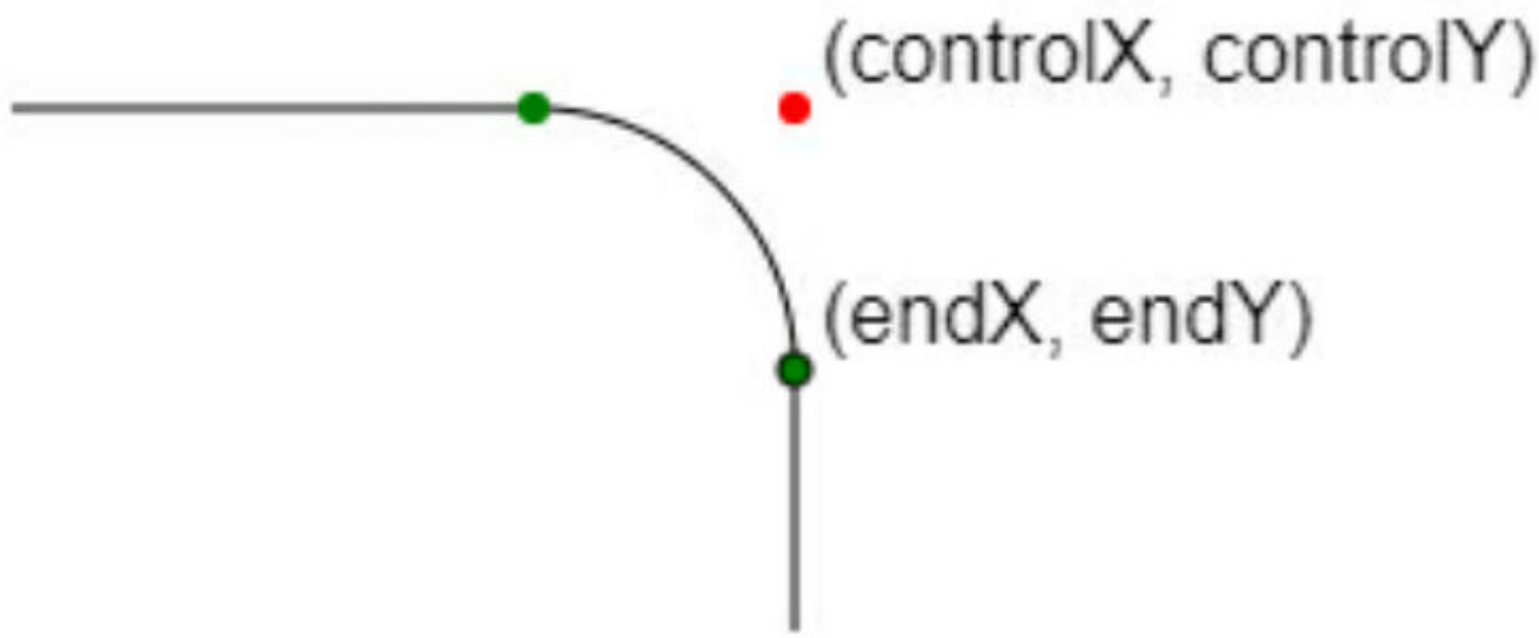
The `arcTo(controlX, controlY, endX, endY, r)` method is used to draw curves between two lines. It takes five arguments:

- the coordinate of the control point `(controlX, controlY)`
- the coordinate of the ending point `(endX, endY)`
- the arc radius (r)

Here's an example that draws an arc between two lines:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    context.beginPath();
    context.moveTo(50, 50);
    context.lineTo(150, 50);
    context.arcTo(200, 50, 200, 100, 50);
    context.lineTo(200, 150);
    context.stroke();
};
```

In this example, we first call the `beginPath()` method to start a new path. Then, we call the `moveTo()` method to move the pen to the starting point of the first line `(50,50)`. Next, we draw the first line using the `lineTo()` method to the point `(150,50)`. Next, we call the `arcTo()` method with the coordinates of the control point and the ending point and the radius of the arc. Finally, we draw the second line using `lineTo()` method and stroke the path with the current stroke style.



9.1.2.3 The quadraticCurveTo method

The quadraticCurveTo() method is another curve-drawing method which uses only one control point to define the direction and amount of curve bending.

The syntax for the quadraticCurveTo(controlX, controlY, endX, endY) method takes four parameters:

- the x-coordinate and y-coordinate of the control point
- the x-coordinate and y-coordinate of the ending point of the curve

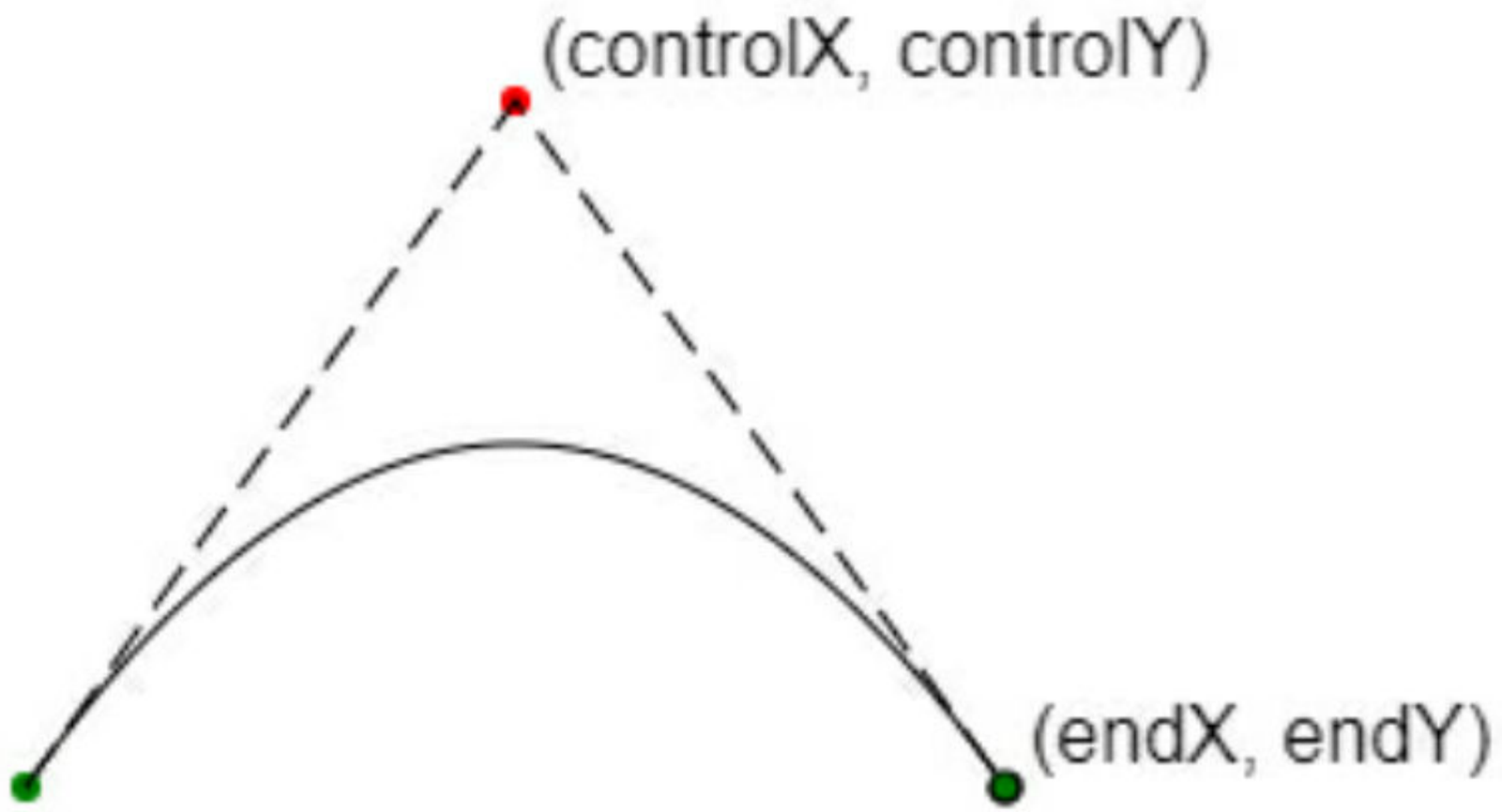
Here is an example of how to use the quadraticCurveTo() method:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");
```

```
    context.beginPath();
    context.moveTo(50, 150);
    context.quadraticCurveTo(150, 10, 250, 150);
    context.stroke();
};
```

In this example, we first call the `beginPath()` method to start a new path and the `moveTo()` method to set the starting point of the curve at `(50, 150)`. Next, we call the `quadraticCurveTo()` method to specify the control point at `(150, 10)` and the end point at `(250, 150)`. Finally, we call the `stroke()` method to draw the curve.

The quadratic curve is a more straightforward curve-drawing method than the Bezier curve, but it can still be used to create smooth and flowing designs in HTML canvas. In addition, by experimenting with different control points and ending points, you can create various curves to add visual interest to your web applications.



9.1.2.4 The bezierCurveTo method

The bezierCurveTo(controlX1, controlY1, controlX2, controlY2, endX, endY) method draws complex curves defined by four control points. Six arguments are required, including:

- the controlX1 and controlY1 coordinates of the first control point
- the controlX2 and controlY2 coordinates of the second control point
- the endX and endY coordinates of the ending point

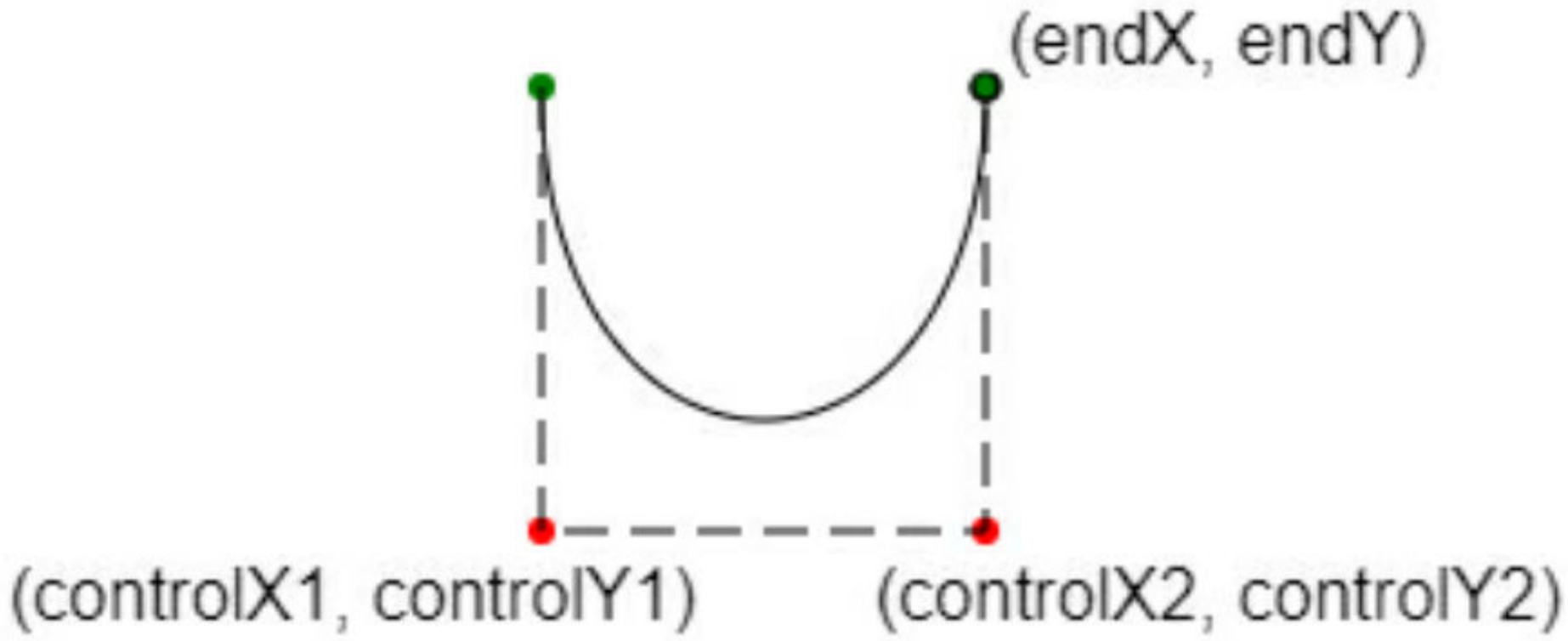
Here's an example that draws a curve using the bezierCurveTo() method:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
```

```
    const context = canvas.getContext("2d");

    context.beginPath();
    context.moveTo(50, 50);
    context.bezierCurveTo(50, 150, 150, 150, 150, 50);
    context.stroke();
};
```

In this example, we first call the `beginPath()` method to start a new path. Then, we call the `moveTo()` method to move the pen to the curve's starting point `(50,50)`. Next, we call the `bezierCurveTo()` method to draw a Bezier curve from the current pen position to the point `(150, 50)` using two control points: `(50, 150)` and `(150, 150)`. Finally, we call the `stroke()` method to stroke the path with the current stroke style.

Overall, curves provide a powerful tool for creating complex shapes and designs in HTML canvas. By understanding how to use Bezier curves and other curve-drawing methods, you can create smooth and flowing designs that add depth and visual interest to your web applications.



9.2 Drawing Shapes

9.2.1 Drawing Rectangles

Drawing rectangles is one of the most basic and valuable tasks in HTML Canvas. Developers can create stunning graphics and visualizations with the canvas element and its API. This section will explore the different ways to draw rectangles in HTML Canvas.

9.2.1.1 The strokeRect() Method

The strokeRect() method draws a rectangular outline with a specific width and height. Here is the format for this method:

```
context.strokeRect(x, y, width, height);
```

Here, " context " is the context object that represents the drawing area, x and y are the coordinates of the rectangle's top-left corner, and width and height are the rectangle's dimensions.

For example, the following code snippet will draw a rectangular outline with a width of 100 and a height of 50, starting from the point (10, 10):

```
window.onload = function () {
   const canvas = document.getElementById("my-canvas");
   const context = canvas.getContext("2d");

   context.strokeRect(10, 10, 100, 50);
};
```

9.2.1.2 The fillRect() Method

The fillRect() method draws a solid-filled rectangle with a specific width and height. The syntax for this method is similar to the strokeRect() method:

```
context.fillRect(x, y, width, height);
```

Here, " context " is the context object that represents the drawing area, x and y are the coordinates of the rectangle's top-left corner, and width and height are the rectangle's dimensions.

For example, the following code snippet will draw a solid-filled rectangle with a width of 100 and a height of 50, starting from the point (10, 10):

```
window.onload = function () {
   const canvas = document.getElementById("my-canvas");
   const context = canvas.getContext("2d");

   context.fillRect(10, 10, 100, 50);
};
```

9.2.1.3 The rect() method with stroke() and fill()

The rect() method is used to define a rectangular path that can be used with the stroke() and fill() methods. Here is the format for this method:

```
context.rect(x, y, width, height);
```

Here, " context " is the context object that represents the drawing area, x and y are the coordinates of the rectangle's top-left corner, and width and height are the rectangle's dimensions.

Once the rectangular path has been defined, it can be filled or stroked using the fill() and stroke() methods, respectively.

For example, the following code snippet will define a rectangular path with a width of 100 and a height of 50, starting from the point (10, 10), and then fill it with a solid color:

```
window.onload = function () {
```

```
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    context.rect(10, 10, 100, 50);
    context.fillStyle = "blue";
    context.fill();
};
```

Alternatively, the following code snippet will define a rectangular path with a width of 100 and a height of 50, starting from the point (10, 10), and then stroke it with a solid color:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    context.rect(10, 10, 100, 50);
    context.strokeStyle = "red";
    context.stroke();
};
```

We can even combine fill() and stroke() methods like this example.

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");
```

```
    context.rect(10, 10, 100, 50);

    context.fillStyle = "blue";
    context.fill();

    context.strokeStyle = "red";
    context.stroke();
};
```

Conclusion

Drawing rectangles in HTML Canvas is fundamental for creating graphics and visualizations. The stro-keRect(), fillRect() , and rect() methods provide developers with different ways to draw rectangular shapes with different styles and effects. By mastering these methods, developers can easily create complex and beautiful graphics.

9.2.2 Drawing Circles

Drawing circles in HTML Canvas is fundamental for creating graphics and visualizations. The arc() method is the most commonly used method for drawing circles in Canvas, but other methods, such as fill() can be used to fill the circle with a color.

To draw a circle in HTML Canvas, we can use the following code:

```
window.onload = function () {
```

```
const canvas = document.getElementById("my-canvas");
const context = canvas.getContext("2d");

const x = canvas.width / 2;
const y = canvas.height / 2;
const radius = 50;

context.beginPath();
context.arc(x, y, radius, 0, 2 * Math.PI);
context.strokeStyle = "blue";
context.fillStyle = "red";
context.stroke();
context.fill();
};
```

In this example, we first set the variables for the center of the circle and its radius. We then begin a new path using the beginPath() method, which is used to start or reset the current path. We then use the arc() method to draw the circle and the stroke() method to stroke the circle with the current stroke style. Additionally, we set the stroke style using the " strokeStyle " property of the context variable. Finally, we also set the fill style using the " fillStyle " property and use the fill() method to fill the circle with the current fill style.

9.2.3 Drawing Gradients

Drawing gradients is a powerful and flexible way to create beautiful graphics and visualizations in HTML Canvas. A gradient is a transition from one color to another, and it can be applied to any shape or object on the canvas. This section will explore different methods for drawing gradients in HTML Canvas.

9.2.3.1 Creating a Linear Gradient

The simplest way to create a gradient in Canvas is to use the createLinearGradient() method. The following code generates a linear gradient that follows a straight path between two points. The method's syntax is shown below:

```
const gradient = context.createLinearGradient(x0, y0, x1, y1);
```

Here, " context " is the variable that represents the drawing area, x0 and y0 are the starting coordinates of the gradient, and x1 and y1 are the ending coordinates of the gradient.

For example, the following code snippet will draw a square with a linear gradient that transitions from red to blue:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");

  const x = 50;
  const y = 50;
```

```
    const width = 200;
    const height = 200;

    const gradient = context.createLinearGradient(x, y, x + width, y + height);
    gradient.addColorStop(0, "red");
    gradient.addColorStop(1, "blue");

    context.fillStyle = gradient;
    context.fillRect(x, y, width, height);
};
```

In this example, we first set the variables for the position and dimensions of the square. We then create a new gradient using the `createLinearGradient()` method and add two color stops using the `addColorStop()` method. Finally, we set the fill style of the context variable to the gradient and use the `fillRect()` method to fill the rectangle with the gradient.

Here's another example of creating a linear gradient with five color stops:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    const x = 50;
    const y = 50;
    const width = 200;
```

```
    const height = 200;

    const gradient = context.createLinearGradient(x, y, x + width, y + height);
    gradient.addColorStop(0, "red");
    gradient.addColorStop(0.25, "orange");
    gradient.addColorStop(0.5, "yellow");
    gradient.addColorStop(0.75, "green");
    gradient.addColorStop(1, "blue");

    context.fillStyle = gradient;
    context.fillRect(x, y, width, height);
};
```

In this example, we use the `createLinearGradient()` method to create a gradient that transitions from red to blue. However, we add three more color stops to create a gradient that transitions through orange, yellow, and green in between. The first argument of the `addColorStop()` method represents the position of the color stop along the gradient, where 0 is the starting point, and 1 is the ending point. The second argument represents the color value. We can create a more complex and colorful gradient by adding multiple color stops.

9.2.3.2 Creating a Radial Gradient

Another way to create a gradient in Canvas is to use the `createRadialGradient()` method. This method creates a radial gradient that follows a circular path from one point to another. Here is the method's syntax:

```
const gradient = context.createRadialGradient(x0, y0, r0, x1, y1, r1);
```

Here, " context " is the variable that represents the drawing area, x0 and y0 are the starting coordinates of the gradient, r0 is the starting radius of the gradient, x1 and y1 are the ending coordinates of the gradient, and r1 is the ending radius of the gradient.

For example, the following code snippet will draw a circle with a radial gradient that transitions from yellow to green:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");

  const x = 150;
  const y = 150;
  const radius = 75;

  const gradient = context.createRadialGradient(x, y, 0, x, y, radius);
  gradient.addColorStop(0, "yellow");
  gradient.addColorStop(1, "green");

  context.fillStyle = gradient;
  context.beginPath();
  context.arc(x, y, radius, 0, 2 * Math.PI);
  context.fill();
```

```
};
```

In this example, we first set the variables for the position and radius of the circle. We then create a new gradient using the createRadialGradient() method and add two color stops using the addColorStop() method. Finally, we set the fill style of the " context " variable to the gradient and use the arc() and fill() methods to draw and fill the circle with the gradient.

9.3 Drawing Text

One of the most common tasks in Canvas is writing text, whether for labelling elements, adding annotations, or creating titles. This section will explore different methods for writing text in HTML Canvas.

9.3.1 Creating Text

The simplest way to create text in Canvas is to use the fillText() or strokeText() methods. These methods accept three arguments: the text to be displayed and the starting point coordinates.

```
context.fillText(text, x, y);
```

and

```
context.strokeText(text, x, y);
```

Here, " context " is the variable that represents the drawing area, the " text " is the string of text to be displayed, and " x " and " y " are the coordinates of the starting point of the text.

For example, the following code snippet will write "Hello, World!" in black text at the position (50, 50) in the canvas:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");
```

```
    context.fillStyle = "black";
    context.font = "24px sans-serif";
    context.fillText("Hello, World!", 50, 50);
};
```

In this example, we first set the fill style of the context variable to black using the " fillStyle " property. We then set the font size and style using the font property. Finally, we use the fillText() method to write the text in the canvas position (50, 50).

Let's explore other versions of the fillText() and strokeText() methods. These methods accept four arguments: the text to be displayed, the starting point coordinates, and maximum width of the text.

```
context.fillText(text, x, y, maxWidth);
```

and

```
context.strokeText(text, x, y, maxWidth);
```

Additionally, the maximum width the text can occupy on the screen is defined as " maxWidth " in pixels. If the text exceeds the maximum width, it will be reduced in size.

For example, the following code snippet will write "Hello, World!" in black text at the position (50, 100) in the canvas, with a maximum width of 100 pixels:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
```

```
    const context = canvas.getContext("2d");

    context.fillStyle = "black";
    context.font = "24px sans-serif";
    context.fillText("Hello, World!", 50, 50);
    context.fillText("Hello, World!", 50, 100, 100);
};
```

In this example, we also use the fillText() method to write a text at the position (50, 50) without a maximum width for references.

9.3.2 Styling Text

9.3.2.1 Text Alignment

In addition to setting the fill style and font properties, we can also style text in Canvas using a variety of properties such as “ textAlign ”, “ textBaseline ”, and “ direction ”.

The “ textAlign ” property sets the alignment of the text relative to its starting position. The possible values are " start ", " end ", " left ", " right ", and " center ". The default value is " start ".

```
context.textAlign = 'center';
```

The “ textBaseline ” property sets the **vertical** alignment of the text relative to its starting position. The possible values are " top ", " hanging ", " middle ", " alphabetic ", " ideographic ", and " bottom ". The default value is " alphabetic ".

```
context.textBaseline = 'middle';
```

For example, the following code snippet will write " Hello, World! " in red text, centered horizontally and vertically:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    context.fillStyle = "red";
    context.font = "24px sans-serif";

    context.textAlign = "center";
    context.textBaseline = "middle";

    context.fillText("Hello, World!", canvas.width / 2, canvas.height / 2);
};
```

In this example, we first set the fill style of the context variable to “ red ” using the “ fillStyle ” property. We then set the font size and style using the “ font ” property. We also set the “ textAlign ” property to " center ", and the “ textBaseline ” property to " middle ". Finally, we use the fillText() method to write the text at the center of the canvas.

9.3.2.2 The font property

The “ context.font ” property is used to set the font and size of the text rendered on the canvas. It is a string that specifies the font style and size and any additional attributes that should be applied to the text. The basic syntax for this property is as follows:

```
context.font = "[font-style] [font-variant] [font-weight] [font-size] [font-family]";
```

Let's break down each component of this syntax:

- **font-style:** Specifies the style of the font, such as “ normal ”, “ italic ”, or “ oblique ”.
- **font-variant:** Specifies the variant of the font, such as “ normal ” or “ small-caps ”.
- **font-weight:** Specifies the font's weight, such as “ normal ”, “ bold ”, “ bolder ”, “ lighter ”, or a number (100, 200 , ..., or 900).
- **font-size:** Specifies the font size in pixels.
- **font-family:** Specifies the font family, such as “ Arial ”, “ Times New Roman ”, or a custom font.

Here are some examples of using the “ font ” property:

```
// Set the font to 20px Arial
context.font = "20px Arial";
```

and

```
// Set the font to bold and italic Times New Roman
context.font = "italic small-caps bold 20px Times New Roman";
```

Here's an example of setting a custom font to the “ context.font ” property:

```
window.onload = function () {
    const canvas = document.getElementById("my-canvas");
    const context = canvas.getContext("2d");

    // Load the custom font
    var customFont = new FontFace("MyFont", "url(path/to/fontfile)");
    customFont.load().then(function (font) {
        document.fonts.add(font);

        // Set the font to the context.font property
        context.font = "bold 50px MyFont, Arial";

        // Render some text using the custom font
        context.fillText("Hello, world!", 50, canvas.height / 2);
    });
};
```

In this example, we use the FontFace API to load a custom font named " MyFont " from a URL. Once the font is loaded, we add it to the “ document.fonts ” object using the add() method.

Next, we assign the “ context.font ” property to employ the personalized font by indicating " MyFont " in the font-family segment of the string, trailed by a substitute font (in this instance, " Arial "). Finally, we use the context.fillText() method to render some text using the custom font.

Not all browsers support the FontFace API, so you may need to provide fallback options or use a font-loading library to ensure your custom fonts are loaded correctly. Additionally, some fonts may have licensing restrictions preventing them from being used on the web, so check the licensing terms before using a custom font in your project.

Once the " context.font " property has been set, any text rendered on the canvas using the fillText() or strokeText() methods will be displayed in the specified font and style. Therefore, if you want to change the font or style of text that will be rendered on the canvas, you will need to call the " context.font " property again with the new settings.

9.4 Drawing Images

Drawing images on HTML canvas is an essential feature that allows you to create dynamic and interactive graphics and visualizations. For example, images can enhance your canvas's appearance, provide context or reference for your drawings, or create animations and games.

The drawImage() method is the primary method for drawing images on a canvas. It allows you to load an image file, create an image object, and then draw that image onto the canvas using various parameters such as position, size, and scaling. With the drawImage() method, you can draw a portion of an image or the entire image onto the canvas and manipulate it to suit your needs.

When working with images on canvas, it's essential to remember that the images need to be loaded and ready to use before you can draw them onto the canvas. This means you'll need to use the Image() con-

structor to create an image object and set its source to the image file you want to use. You can then use the " onload " event handler to ensure the image is loaded before you try to draw it on the canvas.

In addition to the drawImage() method, there are other canvas methods and properties that you can use to manipulate and transform images on the canvas, such as getImageData() and putImageData(), which allow you to manipulate individual pixels in an image. You can also apply filters and effects to images using filters and globalCompositeOperation .

Overall, drawing images on HTML canvas can add depth and interactivity to your web applications, making them more engaging and visually appealing.

9.4.1 drawImage(image, dx, dy)

This drawImage() method allows you to draw an entire image onto a canvas without scaling or cropping it. Here's the syntax for this version of the method:

```
context.drawImage(image, dx, dy);
```

In this syntax, " image " is the object you want to draw onto the canvas, and " dx " and " dy " are the x and y coordinates where you want to place the upper-left corner of the image on the canvas.

Here's an example of using the drawImage() method to draw an image onto a canvas at a specific location:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");
```

```
    const img = new Image();
    img.src = "myImage.png";
    img.onload = function () {
        context.drawImage(img, 50, 50);
    };
};
```

In this example, we create a new " Image " object with the source file "*myImage.png*". We set an " onload " event handler for the image object to ensure it's loaded before we draw it on the canvas. Once the image is loaded, we call the drawImage() method on the context object and pass in the image object and the coordinates where we want to place it on the canvas (50,50).

9.4.2 drawImage(image, dx, dy, dWidth, dHeight)

If you want to draw the image with a specific width and height, you can use the drawImage(image, dx, dy, dWidth, dHeight) method, where " dWidth " and " dHeight " are the desired width and height of the drawn image, respectively. For example:

```
ctx.drawImage(img, 50, 50, 200, 200);
```

This would draw the entire image starting at coordinates (50,50) on the canvas and stretch or shrink it to a size of 200x200 pixels.

Remember that if the image is larger than the canvas, it will be clipped to fit within the canvas bounds. Therefore, to display the entire image, you would need to scale it down to fit within the canvas or use one of the other variations of the drawImage() method to crop or scale the image as needed.

9.4.3 drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)

This variation of the drawImage() method is the most flexible and powerful, allowing you to specify the source image, the source rectangle within the image to be drawn, and the destination rectangle on the canvas where the image should be drawn. Here's the full syntax:

```
context.drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight);
```

In this syntax, " image " is the object you want to draw onto the canvas, " sx " and " sy " are the coordinates of the upper-left corner of the source rectangle in the image, " sWidth " and " sHeight " are the width and height of the source rectangle in the image, " dx " and " dy " are the coordinates of the upper-left corner of the destination rectangle on the canvas, and " dWidth " and " dHeight " are the width and height of the destination rectangle on the canvas.

Here's an example of using the drawImage() method with all nine parameters:

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");

  const img = new Image();
  img.src = "myImage.png";
```

```
img.onload = function () {
    // Draw the middle third of the image, scaled to half size, at (50,50)
    const sx = img.width / 3;
    const sy = 0;
    const sWidth = img.width / 3;
    const sHeight = img.height;

    const dx = 50;
    const dy = 50;
    const dWidth = sWidth / 2;
    const dHeight = sHeight / 2;

    context.drawImage(img, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight);
  };
};
```

In this example, we are drawing the middle third of the image, scaled to half size, at coordinates (50,50) on the canvas. To do this, we set " sx " to the width of the image divided by 3 (which gives us the x coordinate of the start of the middle third), " sy " to 0 (which provides us with the y coordinate of the top of the image), " sWidth " to the width of the image divided by 3 (which gives us the width of the middle third), and " sHeight " to the full height of the image (which provides us with the height of the entire image).

We then set " dx " and " dy " to (50,50) , where we want the upper-left corner of the destination rectangle to be on the canvas. Finally, we set " dWidth " to the width of the middle third of the image divided by 2

(which scales it down to half size) and " dHeight " to the full height of the image divided by 2 (which scales it down to half size as well).

Using this variation of the drawImage() method, you have complete control over which portion of the image you want to draw, how you want to scale it, and where you want to draw it on the canvas. This makes it highly versatile for a wide range of image manipulation tasks.

You can also use the drawImage() method to create animations by drawing different portions of an image on the canvas at different times. For example, you could create a sprite image with multiple frames of an animation and then use the drawImage() method to draw each frame onto the canvas in sequence.

Overall, the drawImage() method is essential for working with images on the HTML canvas. By understanding how to use this method, you can unlock many creative possibilities for your web applications.

9.5 Drawing Shadows

One of the essential visualizations in HTML Canvas is using shadows. Adding shadows to your drawings can help create depth and realism, making your art more visually appealing. This section will explore how to draw shadows in HTML Canvas.

Now, let's move on to drawing shadows in HTML Canvas. To draw a shadow, we first need to specify the shadow properties. We can do this using the shadow properties of the context object.

```
context.shadowColor = "black";
context.shadowBlur = 5;
```

```
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
```

The “ shadowColor ” property sets the color of the shadow. In this example, we set it to “ black ”. The “ shadowBlur ” property sets the blur level of the shadow. A higher value will create a more blurred shadow. Finally, the “ shadowOffsetX ” and “ shadowOffsetY ” properties set the horizontal and vertical distance of the shadow from the object.

Once we have set the shadow properties, we can start drawing our object. Let's draw a simple rectangle as an example.

```
context.fillStyle = "red";
context.fillRect(50, 50, 100, 100);
```

This code will draw a red rectangle with a top-left corner at (50, 50) and a width and height of 100 pixels.

To add a shadow to this rectangle, we need to call the fillRect() method after setting the shadow properties.

```
context.shadowColor = "black";
context.shadowBlur = 5;
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;

context.fillStyle = "red";
```

```
context.fillRect(50, 50, 100, 100);
```

This code will draw the same red rectangle but with a black shadow.

You can also add a shadow to other shapes like circles, polygons, and text.

```
window.onload = function () {
  const canvas = document.getElementById("my-canvas");
  const context = canvas.getContext("2d");

  context.shadowColor = "black";
  context.shadowBlur = 5;
  context.shadowOffsetX = 2;
  context.shadowOffsetY = 2;

  context.beginPath();
  context.arc(150, 150, 100, 0, 2 * Math.PI);
  context.fillStyle = "red";
  context.fill();

  context.shadowColor = "green";
  context.shadowBlur = 10;
  context.shadowOffsetX = 5;
  context.shadowOffsetY = 5;
```

```
    context.font = "bold 24px Arial";
    context.fillStyle = "white";
    context.fillText("Hello World!", 80, 150);
};
```

In this example, we draw a red circle with a black shadow. Then, we change the shadow properties and draw a white "Hello World!" text with a green shadow.

In conclusion, drawing shadows in HTML Canvas is a simple yet powerful technique to add depth and realism to your drawings. By setting the shadow properties of the context object, you can easily create shadows for your shapes and text. Experiment with different shadow properties to achieve the desired effect.

9.6 Clearing Canvas

As a web developer, you may encounter a scenario where you need to remove previously drawn content from an HTML Canvas. This can be achieved using the clearRect() method. This section will explore the clearRect() method and how it can clear the canvas.

The clearRect() method is a built-in HTML Canvas method used to clear a rectangular canvas area. It takes four arguments: x, y, width , and height . These arguments specify the position and size of the rectangular area to be cleared.

```
context.clearRect(x, y, width, height);
```

The “ x ” and “ y ” arguments specify the starting position of the rectangular area to be cleared. The “ width ” and “ height ” arguments specify the rectangle size to be cleared. Here are some examples:

```
// Clear the entire canvas
context.clearRect(0, 0, canvas.width, canvas.height);
```

or

```
// Clear a rectangular area of the canvas
context.clearRect(10, 10, 100, 100);
```

In the first call to clearRect , we clear the entire canvas. In the second call, we clear a rectangular canvas area with a position of (10, 10) and a size of 100 x 100 .

There are many reasons why you should clear the canvas. For example, if you are animating an object, you may need to clear the canvas before redrawing the object in a new position. Likewise, if you are drawing multiple objects on the canvas, you may need to clear the canvas before drawing a new set of objects. Using the clearRect() method, you can remove the previously drawn content from the canvas and start fresh.

CHAPTER 10

HTML Advanced

10.1 Data URIs

The Data URI (Uniform Resource Identifier) enables the inclusion of inline data in web pages, treating them as external resources. The syntax for a data URI is as follows:

```
data:[<media type>][;base64],<data>
```

In this syntax:

- The <media type> (optional) is a MIME type string with the format " type/subtype;parameter-=value ". Its default value is " text/plain;charset=US-ASCII ".
- The optional parameter " ;base64 " indicates that the data is binary and base64-encoded. This parameter is used for media types such as images.

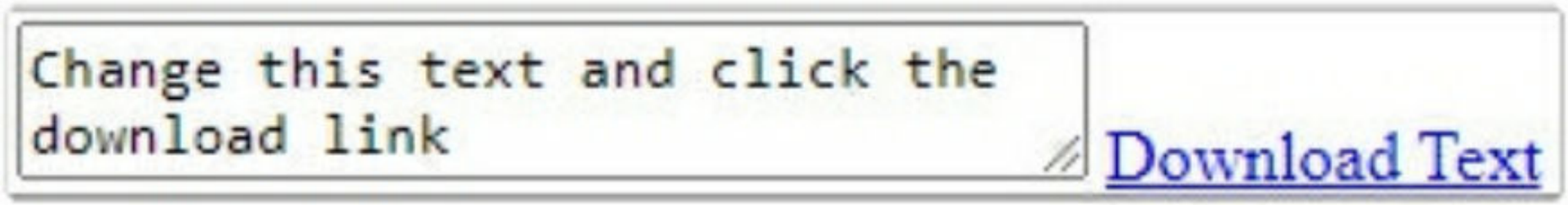
Example:

```
<!DOCTYPE html>
<html>
```

```
<body>
  <textarea name="input" cols="30">Change this text and click the download link</textarea>

  <a href="javascript:void(0)" download="data.txt"> Download Text </a>

  <script type="text/javascript">
    var input = document.querySelector("textarea[ name = 'input' ]");
    var download = document.querySelector("a[ download ]");

    // Listen for input changes so that we can update the HREF
    // attribute of the download link to contain the proper Data URI
    input.addEventListener("input", updateDownloadHref, false);

    // Initialize the download link
    updateDownloadHref();

    function updateDownloadHref() {
      var text = input.value;
      // Build the Data URI
      download.setAttribute("href",  "data:text/plain;charset=utf-8,"  +  encodeURIComponent(
text));
    }
  </script>
</body>
```

```
</html>
```

Result:



10.2 Block Elements

Typically, block elements occupy the entire width of their parent element, regardless of their content. This causes them to be arranged vertically, stacked on each other. You can use CSS properties like width, height, padding , and margin to expand the space of elements beyond their content.

Common HTML block elements include body, header, nav, main, section, footer, h1-h6, p, div, ul, ol, li, pre, blockquote, address, details, figure , and fieldset .

You can use the following CSS declaration to convert different types of elements into block elements:

```
display: block
```

10.3 Inline Elements

Inline elements only occupy the space their content requires without causing line breaks. As a result, they cannot use CSS properties like width, height, padding-top, padding-bottom, margin-top , and mar-

gin-bottom .

Common HTML inline elements include formatting elements (such as b, strong, i, em, mark), label, a, span, code, img, button , and others.

To convert other types of elements to inline elements, we can use the following CSS declaration:

```
display: inline
```

10.4 Inline-block Elements

Inline-block elements combine the features of inline and block elements. They take up only the necessary space for their content without causing line breaks. However, they can have additional space beyond their content using CSS properties such as width, height, padding , and margin .

HTML inline-block elements include button, input, textarea, select, progress, meter, iframe, img, svg, canvas, audio, video , and others.

To convert other types of elements to inline-block elements, we can use the following CSS declaration:

```
display: inline-block
```

CHAPTER 11

HTML Events

HTML events are actions or occurrences that happen on a web page. The user, the browser, or the web page can initiate these events. HTML events provide a way to trigger specific functionality or code when a particular event occurs. Some common examples of HTML events include clicking a button, submitting a form, scrolling the page, or loading the page.

Using event handlers in HTML and JavaScript, developers can create interactive and dynamic web pages that respond to user actions or changes in the page's state. Therefore, understanding HTML events is essential for developing modern, interactive web applications.

11.1 Window Events

11.1.1 onload

The " onload " event is triggered when a web page finishes loading in the browser. Typically, the " onload " event is handled on elements such as <body>, <iframe>, <img>, <link> , and <script> .

Example:

```
<!DOCTYPE html>
<html>
  <body onload="eventHandler()">
    <h1>This is a heading</h1>

    <script>
      function eventHandler() {
        alert("Page is loaded");
      }
    </script>
  </body>
</html>
```

This basic HTML web page includes a JavaScript function and an " onload " event handler.

Inside the <body> element, there is a <script> element that defines a JavaScript function called " eventHandler ". This function displays an alert message that says " Page is loaded ".

The " onload " attribute is used on the <body> element to specify the event that triggers the " eventHandler " function when the page is loaded. The " eventHandler " function is executed when the page is loaded, displaying an alert message that says " Page is loaded ".

11.1.2 onresize

The " onresize " event is triggered when the browser window is being resized.

Example:

```
<!DOCTYPE html>
<html>
  <body onresize="eventHandler()">
    <h1>This is a heading</h1>

    <script>
      function eventHandler() {
        alert("You have changed the size of your browser window!");
      }
    </script>
  </body>
</html>
```

This basic HTML web page includes a JavaScript function and an " onresize " event handler.

Inside the <body> element, there is a <script> element that defines a JavaScript function called " eventHandler ". This function displays an alert message saying, "You have changed the size of your browser window!".

The " onresize " attribute is used on the <body> element to specify the event that triggers the " eventHandler " function when the browser window is resized. When the browser window is resized, the " eventHandler " function is executed, displaying an alert message that says, "You have changed the size of your browser window!".

11.2 Form Events

11.2.1 onfocus

The " onfocus " event is triggered when an element receives focus.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    First name: <input type="text" onfocus="eventHandler(this)" />

    <script>
      function eventHandler(input) {
        input.style.background = "yellow";
      }
    </script>
  </body>
```

```
</html>
```

This basic HTML web page includes a JavaScript function and an " onfocus " event handler.

The input element has a type of " text " and an " onfocus " attribute. The "onfocus" attribute is used to specify the event that triggers the " eventHandler " function when the input element receives focus. When the input element receives focus, the " eventHandler " function is executed with the input element as an argument.

The " eventHandler " function takes the input element as an argument and changes its background color to yellow using the “ style ” property. When the input element receives focus, the " eventHandler " function is executed, changing the background color of the input element to yellow.

11.2.2 onblur

The " onblur " event is triggered when the element no longer has focus.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    Enter your name: <input type="text" id="fname" onblur="eventHandler()" />

    <script>
      function eventHandler() {
```

```
            var x = document.getElementById("fname");
            alert("Hi " + x.value);
        }
    </script>
  </body>
</html>
```

This HTML code creates a form with an input field for entering the user's name. The input field has an id attribute of " fname ". Moreover, the “ onblur ” attribute is applied to the input field to activate the “ eventHandler ” function when the user clicks outside and moves the focus away.

The JavaScript code defines the “ eventHandler ” function that retrieves the input field's value using the “ getElementById ” method to get the input element by its id. Then, it concatenates the string "Hi " with the input field's value and displays an alert dialog box with the resulting message.

Therefore, when the user types their name into the input field and clicks outside of it, the “ eventHandler ” function is called, retrieves the input field's value, concatenates it with "Hi ", and displays an alert dialog box with the resulting message.

11.2.3 onchange

The " onchange " event is fired when the value of an element has been modified. This attribute is supported by HTML tags such as <input type="checkbox">, <input type="file">, <input type="password">, <input type="radio">, <input type="range">, <input type="text">, <select> and <textarea> .

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <select id="my-select" onchange="eventHandler()">
      <option value="Audi">Audi</option>
      <option value="BMW">BMW</option>
      <option value="Mercedes">Mercedes</option>
      <option value="Volvo">Volvo</option>
    </select>

    <script>
      function eventHandler() {
        var x = document.getElementById("my-select").value;
        alert("You selected: " + x);
      }
    </script>
  </body>
</html>
```

This code creates a dropdown menu using the HTML <select> element with four options. The id attribute is " my-select " to uniquely identify the element. The " onchange " attribute is set to " even-

tHandler() ", which means that when the user selects an option from the dropdown menu, the function eventHandler() will be called.

The JavaScript function " eventHandler " obtains the " value " property of the selected option using the " document.getElementById() " method. It then displays an alert box with the message "You selected: " followed by the chosen option's value.

In conclusion, when the user selects an option from the dropdown menu, the eventHandler() function is triggered, showing an alert box containing the selected option's value.

11.2.4 oninput

When a user enters input into an element, the " oninput " event is triggered.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Write some text into the box!</p>

    <input type="text" id="my-input" oninput="eventHandler()" />

    <p id="output"></p>

    <script>
      function eventHandler() {
```

```
        var x = document.getElementById("my-input").value;
        document.getElementById("output").innerHTML = "You wrote: " + x;
      }
    </script>
  </body>
</html>
```

This code creates a simple webpage with an input box and a paragraph element. When the user types something into the box, the " oninput " event is triggered, which calls the " eventHandler() " function defined in the script.

The function gets the value of the input box using the " getElementById() " method and the " value " property of the element. Then it sets the “ innerHTML ” property of the paragraph element with the id " output " to "You wrote: " plus the value typed in the input box.

So, when the user types in the input box, the text they entered is displayed on the webpage as "You wrote: " followed by the text they entered.

11.3 Keyboard Events

11.3.1 onkeydown

The " onkeydown " event is triggered when a user presses a key on the keyboard.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>A function is triggered when the user presses a key in the input field.</p>

    <input type="text" onkeydown="eventHandler()" />

    <script>
      function eventHandler() {
        alert("You pressed a key inside the input field");
      }
    </script>
  </body>
</html>
```

This code creates a webpage with an input field and a paragraph element. The input field has an " onkey-down " attribute that is set to call a JavaScript function named " eventHandler() " whenever a key is pressed inside the input field. The JavaScript function displays an alert box with "You pressed a key inside the input field". This code demonstrates how to use the " onkeydown " event in HTML to execute a function when a user presses a key inside an input field.

11.3.2 onkeyup

The "onkeyup" event is triggered when the user releases a key on the keyboard.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>A function is triggered when the user releases a key in the input field. This function trans-
forms the character to upper case.</p>

    Enter your name: <input type="text" id="fname" onkeyup="eventHandler()" />

    <script>
      function eventHandler() {
        var x = document.getElementById("fname");
        x.value = x.value.toUpperCase();
      }
    </script>
  </body>
</html>
```

A JavaScript function within the HTML code is activated when a user types in an input field. The input field is defined by an HTML <input> tag with type="text" and id="fname" . In addition, the “ onkeyup ” event attribute is set to call the eventHandler() function.

The eventHandler() function gets the input field's value using the document.getElementById() method, then sets the value to its upper case equivalent using the toUpperCase() method. This function will convert any text entered into the input field to an upper case once the user releases a key.

11.3.3 onkeypress

The " onkeypress " event is triggered when a user presses a key on the keyboard.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>A function is triggered when the user presses a key in the input field.</p>

    <input type="text" onkeypress="eventHandler()" />

    <script>
      function eventHandler() {
        alert("You pressed a key inside the input field");
      }
    </script>
  </body>
</html>
```

This HTML code contains an input field and a JavaScript function. The function is triggered when the user presses a key in the input field. Specifically, the “ onkeypress ” attribute calls the eventHandler() function every time a key is pressed inside the input field.

The eventHandler() function displays an alert box with "You pressed a key inside the input field" every time it is called.

11.4 Mouse Events

11.4.1 onclick

The " onclick " event is triggered when the user clicks the element with the mouse.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <button onclick="eventHandler()">Click me</button>

    <script>
      function eventHandler() {
        alert("You clicked the button");
      }
    </script>
```

```
  </body>
</html>
```

This is an HTML document with a button and a JavaScript function. The JavaScript function called eventHandler() is executed when the button is clicked.

The eventHandler() function contains an alert() method, which displays a popup window with the message "You clicked the button". Therefore, when clicking the button, the user will see a popup window with this message.

11.4.2 ondblclick

The " ondblclick " event is triggered when the mouse double-clicks the element.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <button ondblclick="eventHandler()">Click me</button>

    <script>
      function eventHandler() {
        alert("You double-clicked the button");
      }
    </script>
```

```
  </body>
</html>
```

This HTML code creates a button element with the " ondblclick " attribute that specifies a JavaScript function to be executed when the button is double-clicked.

The JavaScript function " eventHandler() " is defined in the script section of the page. When you double-click the button, the function will be triggered, and you will see an alert dialog box that says, "You double-clicked the button".

11.4.3 onmousemove

The " onmousemove " event is triggered when the cursor is moved while it is on top of an element.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p onmousemove="enlarge(this)">The function enlarge() is triggered when the user mouse pointer is moved over the paragraph. This function will enlarge the font size.</p>

    <script>
      function enlarge(x) {
        var oldFontSize = x.style["font-size"];
        if (!oldFontSize) {
```

```
            oldFontSize = 16;
        } else {
            oldFontSize = parseFloat(oldFontSize.replace("px", ""));
        }

        x.style["font-size"] = oldFontSize + 0.1 + "px";
    }
  </script>
 </body>
</html>
```

This code defines an HTML document that contains a paragraph element with an " onmousemove " attribute set to call a function called " enlarge " when the user's mouse pointer is moved over it. The " enlarge " function requires an argument " x " that identifies the specific element that initiated the event.

The function first gets the current font size of the element, and if it is not defined, it sets the font size to 16. Otherwise, it converts the font size from a string to a number, removes the " px " unit, and then adds 0.1. The new font size is then set as the value of the " font-size " CSS property of the element, causing the font to appear larger.

11.4.4 onmousedown and onmouseup

The " onmousedown " event is triggered when the user presses a mouse button over the element. Likewise, the " onmouseup " event is triggered when the user releases the mouse button over the element.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p id="p1" onmousedown="mouseDown()" onmouseup="mouseUp()">
      Click this text!<br />
      The mouseDown() function is triggered when the mouse button is pressed over this paragraph.
The function sets the color of the text to red. <br />
      The mouseUp() function is triggered when the mouse button is released while over the para-
graph. The function sets the color of the text to blue.
    </p>

    <script>
      function mouseDown() {
        document.getElementById("p1").style.color = "red";
      }

      function mouseUp() {
        document.getElementById("p1").style.color = "blue";
      }
    </script>
  </body>
```

```
</html>
```

This HTML code creates a paragraph element with id " p1 " and defines two event handlers for it using the " onmousedown " and " onmouseup " attributes. The first event handler, " onmousedown ", is triggered when the user presses a mouse button down while over the paragraph element, and it calls the " mouseDown() " function, which sets the color of the paragraph text to red.

The second event handler, " onmouseup ", is triggered when the user releases the mouse button while still over the paragraph element, and it calls the " mouseUp() " function, which sets the color of the paragraph text to blue. Thus, when the user clicks on the paragraph and then releases the mouse button, the text color will change from red to blue.

11.4.5 onmouseover and onmouseout

The " onmouseover " event is triggered when the mouse pointer moves over an element, whereas the " onmouseout " event is triggered when the mouse pointer moves out of an element.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p onmouseover="increaseFontSize(this)" onmouseout="decreaseFontSize(this)">
      When the user mouse over the paragraph, its font size will be 32px. <br />
      When the mouse pointer is moved out of it, the font size will be 20px.
```

```
    </p>

    <script>
      function increaseFontSize(ele) {
        ele.style["font-size"] = "32px";
      }

      function decreaseFontSize(ele) {
        ele.style["font-size"] = "20px";
      }
    </script>
  </body>
</html>
```

This code creates a paragraph element that changes its font size when the user moves the mouse over or out of it.

The " onmouseover " attribute is used to call the increaseFontSize() function when the user moves the mouse over the paragraph element. This function takes in the " ele " parameter, the paragraph element itself. The increaseFontSize() function then sets the font size of the paragraph element to 32px.

Similarly, the " onmouseout " attribute is used to call the decreaseFontSize() function when the user moves the mouse out of the paragraph element. This function also takes in the " ele " parameter, the para-

graph element itself. The decreaseFontSize() function then sets the font size of the paragraph element to 20px.

11.5 Clipboard Events

11.5.1 oncopy

The " oncopy " event is triggered when the content of an element is copied by the user, which can be done in two ways:

- Pressing CTRL + C
- Right-click and select the "Copy" command from the context menu

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <input type="text" oncopy="eventHandler()" value="Try to copy this text" />

    <script>
      function eventHandler() {
        alert("You copied the text.");
      }
    </script>
```

```
  </body>
</html>
```

This simple HTML code includes an input field of type " text " and an " oncopy " event handler. The input field initially displays "Try to copy this text".

The " oncopy " event handler is triggered when the user copies the input field's content. When the " oncopy " event is triggered, it calls the " eventHandler() " function defined in the <script> tag. This function displays an alert message saying, "You copied the text".

11.5.2 oncut

The “ oncut ” event is triggered when the user cuts the content of an element. There are two ways to cut the content:

- Pressing CTRL + X
- Right-click and select the "Cut" command from the context menu

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <input type="text" oncut="eventHandler()" value="Try to cut this text" />

    <script>
```

```
        function eventHandler() {
            alert("You cut the text.");
        }
    </script>
  </body>
</html>
```

This HTML code contains an input element of type " text " with the "Try to cut this text" value. In addition, the " oncut " attribute is added to this input element with the value " eventHandler() ".

The " oncut " attribute defines what happens when a user cuts the content of an input element. The " eventHandler() " function is called when the " oncut " event is triggered.

The JavaScript code defines the " eventHandler() " function, which displays an alert message "You cut the text." when the " oncut " event is triggered.

So, when the user cuts the text inside the input element, the " eventHandler() " function will be called and display an alert message.

11.5.3 onpaste

The " onpaste " attribute triggers an event when the user pastes content into an element. Two ways to paste the content into an element are:

- Pressing CTRL + V
- Right-click and select the "Paste" command from the context menu

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <input type="text" onpaste="eventHandler()" value="Try to paste text in here" size="40" />

    <script>
      function eventHandler() {
         alert("You pasted the text.");
      }
    </script>
  </body>
</html>
```

This code displays a text input element in the body of an HTML document with an initial value "Try to paste text in here". The “ onpaste ” attribute is added to the input element, which calls the eventHandler() function whenever the user pastes some text into the input element.

The eventHandler() function is defined in the <script> tag. It displays an alert with "You pasted the text." whenever it is called.

Therefore, when the user pastes text into the input element, the eventHandler() function will be called, and an alert message will be displayed.

CHAPTER 12

HTML APIs

HTML APIs (Application Programming Interfaces) are a set of interfaces, protocols, and tools for building web applications using HTML, CSS, and JavaScript. These APIs provide developers with standardized ways to interact with various web browser features and functionality, such as manipulating the browser's Document Object Model (DOM), accessing device hardware and software features, performing network requests, and more.

Many HTML APIs are available, and each provides a specific set of functionality that developers can use to create richer, more interactive web applications. Some of the most commonly used HTML APIs include:

- **DOM API**: The Document Object Model API allows developers to dynamically manipulate and interact with the elements on a web page. Developers can use this API to create, remove, and modify elements on the page and handle events such as mouse clicks, keyboard presses, and form submissions.
- **Geolocation API**: This API allows developers to retrieve a user's location data (latitude and longitude) using the device's GPS or other location-based services. This is useful for creating location-aware applications like maps, weather apps, and social media platforms.

- **Web Storage API**: This API allows developers to store data in the user's web browser, allowing for persistent data between page refreshes and even after the user closes the browser. This is useful for creating applications that store user preferences, shopping cart data, and other persistent data.
- **XMLHttpRequest API**: This API allows developers to make HTTP requests from the client-side JavaScript code, allowing web applications to communicate with web servers and retrieve data in real-time without needing to refresh the page. This is essential for building dynamic, responsive applications that update and display real-time data.
- **Web Worker API**: Web Workers allow scripts to be executed in a dedicated background thread that runs separately from the main thread. This API allows offloading complex and time-consuming tasks from the main thread, making the user interface more responsive. With Web Workers, scripts can run in the background without blocking the UI, providing a smoother user experience.

These are just a few examples of the HTML APIs available to developers. As web technologies evolve, new APIs are constantly being developed and added to the HTML specification, providing developers with even more tools and functionality to create rich, interactive web applications.

12.1 DOM APIs

The DOM API provides a way to manipulate and interact with the elements and content of an HTML document. DOM Manipulation refers to adding, removing, or modifying elements.

There are several methods provided by the DOM API to manipulate the DOM. Some of the most commonly used methods are mentioned below.

12.1.1 Selecting HTML Elements

12.1.1.1 getElementById()

In the DOM API, getElementById() is a method that allows you to select an HTML element based on its unique identifier, which is defined using the " id " attribute in the HTML code. Below is an example.

Let's say you have the following HTML code:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <p id="my-paragraph">Hello, World!</p>

    <script src="app.js"></script>
  </body>
</html>
```

To select the paragraph element with an id of “ my-paragraph ”, you can use the getElementById() method in JavaScript like this:

app.js

```
let paragraph = document.getElementById("my-paragraph");
console.log(paragraph.textContent);
```

This code selects the paragraph element with an id of “ my-paragraph ” and assigns it to a variable called “ paragraph ”. Then, it logs the value of the “ textContent ” property of the paragraph element to the console using the console.log() method. The “ textContent ” property contains the element's text content, including any HTML tags within it.

You can then manipulate this element using other DOM API methods or properties. For example, suppose that you want to change the text of the paragraph to "Have a nice day". You can do it using the “textContent” property like this:

```
paragraph.textContent = "Have a nice day";
```

Now, the paragraph element will display the text "Have a nice day" instead of "Hello, World!".

12.1.1.2 getElementsByClassName()

The getElementsByClassName() method retrieves a collection of elements with the same class name. It accepts a single argument: the class name of the elements you want to retrieve and returns a collection

of elements that match the specified class name. If no elements match the class name, an empty array is returned.

Here is an example of how to use getElementsByClassName() :

```
<!DOCTYPE html>
<html>
  <body>
    <div class="my-class">Element 1</div>
    <div class="my-class">Element 2</div>
    <div class="my-class">Element 3</div>

    <script>
      let elements = document.getElementsByClassName("my-class");
      console.log(elements);
    </script>
  </body>
</html>
```

In this example, we have three elements with the class name of " my-class ". We use document.getElementsByClassName("my-class") to retrieve a collection of all the elements with that class name. The collection is stored in the " elements " variable. Finally, we log the " elements " array to the console.

After running this code in the browser, the console should display the output described below:

```
HTMLCollection(3) [div.my-class, div.my-class, div.my-class]
```

This output shows that the getElementsByClassName() method returned a collection of three elements that match the “ my-class ” class name.

12.1.1.3 getElementsByTagName()

The getElementsByTagName() method is part of DOM API. It allows developers to retrieve an HTML collection of all elements with a particular tag name within the current document, which can then be manipulated using JavaScript.

The syntax for using getElementsByTagName() is as follows:

```
document.getElementsByTagName(tagName);
```

where “ tagName ” is a string representing the name of the HTML tag you want to retrieve. For example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Heading</h1>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
```

```
    <script src="app.js"></script>
  </body>
</html>
```

To retrieve all of the <p> elements on the page, you would use the following JavaScript code:

app.js

```
let paragraphs = document.getElementsByTagName("p");
console.log(paragraphs);
```

This would return an HTML collection of all <p> elements on the page. You could then access and manipulate each element individually using its index in the collection, like this:

```
paragraphs[0].textContent = "New text for the first paragraph";
```

In this example, we are accessing the first paragraph element in the collection using its index (0), then setting its " textContent " property to a new value.

Note that getElementsByTagName() returns an HTML collection, not an array, so you cannot use array methods like forEach() directly on it. However, you can convert it to an array using Array.from() or the spread operator, like this:

```
let paragraphsArray = Array.from(paragraphs);
//// or
// let paragraphsArray = [...paragraphs];
```

```
paragraphsArray.forEach(function (paragraph) {
  paragraph.style.color = "red";
});
```

In this example, we first convert the paragraphs collection to an array using `Array.from()`, then using the `forEach()` method to iterate over each element in the array and set its color style to red.

12.1.1.4 querySelector()

With the DOM API method `querySelector()`, you can choose the first element that matches your specific CSS selector. It will return `null` if there is no such element.

Here is an example of using `querySelector()`:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .highlight {
        color: red;
        font-weight: bold;
      }
    </style>
```

```
  </head>
  <body>
    <h1>Heading</h1>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>

    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
// Select the first element with the tag name "p"
let highlighted = document.querySelector("p");

// Add a new class to the selected element
highlighted.classList.add("highlight");
```

In this example, we use querySelector() to select the first element with the tag name " p " and store it in the " highlighted " variable. We then add a new class " highlight " to the selected element using classList.add() .

12.1.1.5 querySelectorAll()

querySelectorAll() is a method in the DOM API that allows you to select multiple elements in a docu-

ment using a CSS selector. It returns a NodeList of elements that match the selector or an empty NodeList if no elements match.

Here's an example of using querySelectorAll() :

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Heading</h1>
    <p class="text">Paragraph 1</p>
    <p class="text">Paragraph 2</p>

    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
const paragraphs = document.querySelectorAll(".text");

paragraphs.forEach((paragraph) => {
  paragraph.style.color = "red";
});
```

In this example, we have two paragraphs with the class " text ". The JavaScript code selects both paragraphs using the “ .text ” selector and sets their color to red using the “ style.color ” property. The queryS-

electorAll() method returns a NodeList that contains both paragraphs, which we can loop through using the forEach() method.

12.1.2 Creating, Adding or Removing Elements

12.1.2.1 createElement()

In the DOM API, createElement() is a method to create a new HTML element. It takes the tag name as a parameter and returns a new HTML element object.

Here's an example:

```
const newButton = document.createElement("button");
```

This code creates a new button element and assigns it to a variable called " newButton ". The newly created button only exists in the HTML document once we add it using other DOM methods like appendChild() .

Here's an example that creates a new button element, sets some of its attributes, and appends it to an existing element:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Heading</h1>
```

```
    <div id="container">
      <p class="text">Paragraph 1</p>
      <p class="text">Paragraph 2</p>
    </div>

    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
const newButton = document.createElement("button");
newButton.textContent = "Click me!";
newButton.setAttribute("id", "my-button");
newButton.setAttribute("class", "primary-button");

const containerDiv = document.getElementById("container");
containerDiv.appendChild(newButton);
```

In this code, we create a new button element using createElement("button") . Then, we set some of its attributes using " textContent " and setAttribute() . Finally, we select an existing element using getElementById() and append the new button using appendChild() . The result will be a new button with the text "Click me!" and the id and class attributes set to " my-button " and " primary-button ", respectively, appended to the div with the id " container ".

12.1.2.2 appendChild()

In the DOM API, the `appendChild()` method is used to append a child element to a parent element. It takes a single parameter, the child element, to be added to the parent.

In the previous section, we learned how to add a new element to an existing one using the `appendChild()` method.

12.1.2.3 removeChild()

The `removeChild()` method removes a child node from a parent element. This method is used to manipulate HTML elements dynamically using JavaScript.

Here's an example of how to use `removeChild()`:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <div id="my-div">
      <p>First paragraph</p>
      <p>Second paragraph</p>
      <p>Third paragraph</p>
    </div>
```

```
    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
const parent = document.getElementById("my-div");

// selects the second paragraph element
const child = parent.querySelector("p:nth-child(2)");

// removes the second paragraph element
parent.removeChild(child);
```

In this example, the element with the ID "my-div" contains three <p> elements. We use the getElementById() method to select the parent element and the querySelector() method to select the second paragraph element as the child to be removed. Finally, we call the removeChild() method on the parent element and pass the child element as a parameter to remove it.

Note that removeChild() only works on child nodes that are part of the parent element. If you try to remove a node that is not a child of the parent, you will get an error.

12.1.3 Manipulating HTML Elements

HTML elements form the foundation of an HTML document. Each element has unique attributes and

properties that define how the content is presented on the web page. In this section, we will compare the attributes and properties of HTML elements.

Attributes of HTML Elements

Additional information about an element can be provided by using HTML attributes. Attributes are included in the opening tag of an element and are written as name-value pairs. Here are some common attributes of HTML elements:

- **Class**: The " class " attribute specifies one or more CSS classes to an element. CSS classes allow you to apply styles to multiple elements at once.
- **ID**: The " id " attribute declares a unique identifier for an element. The ID is used to identify an element in CSS and JavaScript.
- **Style**: The " style " attribute specifies inline styles for an element. Inline styles are applied directly to the element and override any styles specified in CSS.
- **Title**: The " title " attribute provides additional information about an element. The title text is displayed as a tooltip when a user hovers over the element.
- **Href**: The " href " attribute specifies the URL of a link. It is used in the `<a>` tag to create hyperlinks.

Properties of HTML Elements

HTML properties are used to set or retrieve the values of an element. Properties are accessed through JavaScript and are used to manipulate the content and behavior of an element. Here are some common properties of HTML elements:

- **InnerHTML**: The " innerHTML " property is used to set or retrieve the content of an element. The content can be text, HTML, or a combination of both.
- **Value**: The " value " property is used to set or retrieve the value of an input element. It is used in input, select , and textarea elements.
- **Src**: The " src " property is used to specify the URL of an image or media file. It is used in the img, audio , and video elements.
- **Disabled**: The " disabled " property is used to disable an input element. When an element is disabled, it cannot be edited or clicked.
- **Checked**: The " checked " property checks or unchecks a checkbox or radio button.

* * *

Comparison of Attributes and Properties

Attributes and properties of HTML elements have some differences. Here are some comparisons between the two:

- Attributes are initialized in the HTML code, while properties are accessed through JavaScript.
- Properties are utilized for modifying an HTML element's content and behavior; as a result, there are typically more properties than attributes.

12.1.3.1 hasAttribute()

The hasAttribute() method checks if an element has a specific attribute. It returns true if the element has the attribute and false if it doesn't.

Here's an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <div id="my-div" my-data-type="example">This is a div element</div>

    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
const myDiv = document.getElementById("my-div");

if (myDiv.hasAttribute("my-data-type")) {
  console.log("The 'my-data-type' attribute exists");
} else {
```

```
    console.log("The 'my-data-type' attribute does not exist");
}
```

We have a element with an attribute " my-data-type " in this example. The JavaScript code uses the getElementById() method to get a reference to the element and then uses the hasAttribute() method to check if it has the " my-data-type " attribute.

Since the attribute does exist, the code will output "The 'my-data-type' attribute exists" to the console.

12.1.3.2 getAttribute()

The getAttribute() method is a DOM API method to retrieve the value of a specified attribute on an HTML element. It takes one argument: the name of the attribute you want to retrieve.

Here is an example:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <div id="my-div" data-color="blue">This is a div element</div>

    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
const myDiv = document.getElementById("my-div");

const colorAttribute = myDiv.getAttribute("data-color");
console.log(colorAttribute); // Output: "blue"
```

In this example, we have a element with an id of " my-div " and a custom “ data-color ” attribute with a value of " blue ". We then use the document.getElementById() method to obtain the element, and use the getAttribute() method to retrieve the value of the “ data-color ” attribute. The value is then logged to the console using console.log() . The output in the console will be the string "blue".

12.1.3.3 setAttribute()

With JavaScript's DOM API, you can use the setAttribute() method to set the value of an attribute for an element.

The syntax for setAttribute() is as follows:

```
element.setAttribute(name, value);
```

where “ element ” is the DOM element you want to modify, “ name ” is the name of the attribute you want to set, and “ value ” is the value you want to assign to that attribute.

Here's an example of how to use setAttribute() :

```
const image = document.getElementById("my-image");
```

```
image.setAttribute("src", "example.jpg");
```

In this example, the code selects the image element with an ID of " my-image " and sets its " src " attribute to " example.jpg ".

You can also use setAttribute() to create new attributes on an element that didn't previously exist. For example:

```
const image = document.getElementById("my-image");
image.setAttribute("data-description", "A beautiful day");
```

In this example, the code adds a new attribute called " data-description " to the image element and assigns it the value "A beautiful day".

12.1.3.4 Using Properties to Manipulate Elements

Properties are the characteristics of an HTML element that define its behavior and appearance. To access an element's properties with JavaScript, you must first select the element using the Document Object Model (DOM). After choosing the element, you can use the dot notation to access its properties.

For example, we will manipulate the style of HTML elements using the “ style ” and “ classList ” properties as below.

```
element.style.property
```

This syntax allows you to set a specific CSS property on an element. If you want to make the background color of a paragraph blue, you can do the following:

```
const para = document.querySelector("p");
para.style.backgroundColor = "blue";
```

```
element.classList.add(className)
```

This syntax adds a CSS class to an element's class list. For example, to add a class of ' highlight ' to a paragraph:

```
const para = document.querySelector("p");
para.classList.add("highlight");
```

```
element.classList.remove(className)
```

This syntax removes a CSS class from an element's list of classes. For example, to remove the ' highlight ' class from a paragraph:

```
const para = document.querySelector("p");
para.classList.remove("highlight");
```

```
element.classList.toggle(className)
```

This syntax adds a class to an element if it doesn't have it, or removes it if it does. For example, to toggle a ' highlight ' class on and off for a paragraph:

```
const para = document.querySelector("p");
para.classList.toggle("highlight");
```

Above are some examples of manipulating HTML elements by using their properties.

12.1.4 Drag and Drop

JavaScript code allows us to implement drag-and-drop functionality for HTML elements. First, we set an element's " draggable " attribute to true to make an element draggable.

```
<div id="div1" draggable="true">This div is draggable</div>
```

Then, we can specify what happens when the element is dragged using the " ondragstart " attribute, which invokes a callback function that sets the dragged data using the " dataTransfer.setData() " method. Here's an example:

```
<div id="div1" draggable="true" ondragstart="dragStartHandler(event)">This div is draggable</div>

<script>
  function dragStartHandler(evt) {
     evt.dataTransfer.setData("sourceID", evt.target.id);
  }
</script>
```

To enable a drop zone for the dragged element, we must prevent the default handling of the HTML elements by using the " ondragover " attribute:

```
<div id="div2" ondragover="allowDropping(event)"></div>

<script>
  function allowDropping(evt) {
    evt.preventDefault();
  }
</script>
```

When the dragged element is dropped onto the drop zone, a drop event occurs, and the " ondrop " attribute invokes a callback function that retrieves the dragged data (the dragged element's ID) using the " dataTransfer.getData() " method. We can then get the dragged element by the ID and append it to the drop zone. Here's an example:

```
<div id="div2" ondrop="dropHandler(event)" ondragover="allowDropping(event)"></div>

<script>
  function dropHandler(ev) {
    var data = ev.dataTransfer.getData("sourceID");
    ev.target.appendChild(document.getElementById(data));
  }
</script>
```

This is the final code:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      #div2 {
        border: 1px solid black;
        width: 200px;
        height: 60px;
      }
    </style>
  </head>
  <body>
    <div id="div1" draggable="true" ondragstart="dragStartHandler(event)">This div is draggable. You can drag and drop it into the rectangle.</div>

    <div id="div2" ondrop="dropHandler(event)" ondragover="allowDropping(event)"></div>

    <script>
      function allowDropping(ev) {
        ev.preventDefault();
      }
```

```
        function dropHandler(ev) {
            var data = ev.dataTransfer.getData("sourceID");
            ev.target.appendChild(document.getElementById(data));
        }

        function dragStartHandler(ev) {
            console.log(ev);
            ev.dataTransfer.setData("sourceID", ev.target.id);
        }
    </script>
</body>
</html>
```

12.2 Geolocation API

The Geolocation API in JavaScript allows us to retrieve the user's current geographical position, which includes their latitude and longitude. For privacy reasons, the user must approve using this feature before accessing the position information.

To obtain the user's current location, we can use the getCurrentPosition() method of the " navigator. geolocation " object. This function takes a callback function as its parameter, which receives a coordinate object containing the latitude and longitude of the user's position. For example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Click the button to get your coordinates.</p>

    <button onclick="getLocation()">Get location</button>

    <p id="out"></p>

    <script>
      var x = document.getElementById("out");

      function getLocation() {
        if (navigator.geolocation) {
          navigator.geolocation.getCurrentPosition(showPosition);
        } else {
          x.innerHTML = "Geolocation is not supported by this browser.";
        }
      }

      function showPosition(position) {
        x.innerHTML = "Latitude: " + position.coords.latitude + "<br>Longitude: " + position.coords.
longitude;
      }
```

```
    </script>
  </body>
</html>
```

Within the body tag, a paragraph prompts the user to click a button to get their coordinates. It is followed by a button with an “ onclick ” attribute that calls the getLocation() function when clicked.

Below the button is a paragraph element with an id of " out ", which is an empty placeholder to display the user's coordinates.

In the <script> tag, the getLocation() function is defined. It first checks if the user's browser supports the Geolocation API. If it does, the getCurrentPosition() method is called, which takes a callback function called showPosition() as its parameter.

If the browser does not support the Geolocation API, the “ x.innerHTML ” property is set to "Geolocation is not supported by this browser."

The showPosition() function takes a position parameter containing the user's coordinates. The latitude and longitude coordinates are then extracted from the “ position ” object and displayed in the “ out ” paragraph element using the “ innerHTML ” property.

Overall, when the user clicks the "Get location" button, the getLocation() function is called, checking if the Geolocation API is supported. If it is, it calls the showPosition() function to display the user's coordinates.

12.3 Web Storage API

In the past, web developers used cookies to store application data, but this approach harmed website performance because cookies were included in every server request.

However, with the introduction of HTML5, web storage was introduced as an alternative. Storing data locally in the browser using web storage is more secure, allows for more significant amounts of data to be stored (up to 5MB), and does not transfer data to the server.

Before using web storage, it is important to check if the browser supports it first:

```
if (typeof Storage !== "undefined") {
  // Code for Web Storage
} else {
  // Not support Web Storage
}
```

On the client side, HTML web storage offers two objects that can be used to store data:

- localStorage : stores data with no expiration date
- sessionStorage : the data is only stored for one session and will be lost once the browser tab is closed

12.3.1 The localStorage Object

Data stored in localStorage is persistent and will remain even after the browser is closed. We can store and retrieve data using the setItem() and getItem() methods.

For instance, we can store a person's first name by calling the setItem() method as follows:

```
// Store
localStorage.setItem("firstname", "John");
```

To retrieve the stored data, we can use the getItem() method:

```
// Retrieve
var firstname = localStorage.getItem("firstname");
```

Alternatively, we can use the dot notation to store and retrieve data, as shown below:

```
// Store
localStorage.firstname = "John";

// Retrieve
var firstname = localStorage.firstname;
```

To remove an item from the localStorage , we can use the removeItem() method:

```
localStorage.removeItem("firstname");
```

It is worth noting that all data stored in localStorage is in the form of strings, and we need to convert them to other formats when necessary.

Here is an example of how localStorage works in practice.

```
<!DOCTYPE html>
<html>
  <body>
    <p>Step 1. Enter your name:</p>
    <input id="name" type="text" />

    <p>Step 2. Click this button to set it to the Local Storage:</p>
    <input type="button" value="Set data" onclick="setItem()" />

    <p>Step 3. Open this webpage in another tab.</p>

    <p>Step 4. Click this button to get data from the Local Storage:</p>
    <input type="button" value="Get data" onclick="getItem()" />

    <p>Step 5. Close all tabs and open this page again. Then, click the "Get data" button to see if we can still get the data.</p>
```

```
<script>
  function setItem() {
    if (typeof Storage !== "undefined") {
      var name = document.getElementById("name").value;
      localStorage.myName = name;
    }
  }

  function getItem() {
    if (typeof Storage !== "undefined") {
      alert(localStorage.myName);
    }
  }
</script>
</body>
</html>
```

This simple HTML web page demonstrates how to use localStorage to store and retrieve data. The web page contains two buttons allowing users to set and get data from the localStorage .

The first button, labelled "Set data", calls the JavaScript function setItem() when clicked. This function checks if the browser supports localStorage , retrieves the name entered by the user in the input field with the id " name ", and stores it in the localStorage with the key " myName ".

The second button, labelled "Get data", calls the JavaScript function getItem() when clicked. This function also checks if the browser supports localStorage and retrieves the value stored with the key " myName " in the localStorage . The retrieved value is then displayed in an alert message.

The web page also includes some instructions for the user to follow to test the web page's functionality. For example, please enter your name, set the data to the localStorage , open the page in another tab, retrieve the data from the localStorage , close all tabs, and reopen the page to see if the data is still accessible.

Overall, this code demonstrates how to use localStorage to store and retrieve data on a web page, allowing the data to persist even after the user closes the browser.

12.3.2 The sessionStorage Object

The sessionStorage object is comparable to the localStorage object, except it stores data for just one session. As a result, when users close the tab or window of the browser, the data will be deleted.

Example:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Step 1. Enter your name:</p>
    <input id="name" type="text" />

    <p>Step 2. Click this button to set it to the Session Storage:</p>
    <input type="button" value="Set data" onclick="setItem()" />
```

```
    <p>Step 3. Click this button to get data from the Session Storage:</p>
    <input type="button" value="Get data" onclick="getItem()" />

    <p>Step 4. Open this webpage in another tab.</p>

    <p>Step 5. Then, click the "Get data" button to see that we cannot get the data since it's only available in the first tab.</p>

    <script>
      function setItem() {
        if (typeof Storage !== "undefined") {
          var name = document.getElementById("name").value;
          sessionStorage.myName = name;
        }
      }

      function getItem() {
        if (typeof Storage !== "undefined") {
          alert(sessionStorage.myName);
        }
      }
    </script>
</body>
```

```
</html>
```

12.4 XMLHttpRequest (XHR) API

XMLHttpRequest (XHR) API is a web technology that enables client-side scripts to communicate with a server-side script asynchronously. As a result, it allows web pages to update content without requiring a full page reload, providing a smoother and more responsive user experience. This section will dive into the XMLHttpRequest API and its various features.

The XMLHttpRequest API is a powerful tool for making HTTP requests from the client side. It's also helpful in requesting APIs or web services and sending or receiving data in various formats, such as JSON and XML. The API consists of several methods and properties that can be used to customize requests and responses.

12.4.1 Creating an XHR Object

Before making a request using the XHR API, we must create an instance of the XMLHttpRequest object. We can do this as the below code:

```
const xhr = new XMLHttpRequest();
```

The above code creates a new instance of the XMLHttpRequest object and assigns it to the variable " xhr ".

12.4.2 Making and Sending a Request

Once we have an XHR object, we can use it to make a request to a server. To do this, we must call the open() method and specify the HTTP method and the URL of the resource we want to request:

```
xhr.open('GET', 'https://example.com/data.json');
```

In the above code, we make a GET request to the URL "*https://example.com/data.json*".

After we have opened the request, we need to send it using the send() method:

```
xhr.send();
```

In the above code, we are sending the request without any data.

12.4.3 Handling the Response

Once we have sent the request, we need to handle the response. This can be done using the " onload " event, triggered when the server responds to the request successfully with a status code of 200 (OK).

The response can be accessed using the " responseText " property of the XHR object:

```
xhr.onload = function() {
  console.log(xhr.responseText);
};
```

In the above code, we log the response text to the console.

12.4.4 Handling Errors

Sometimes, the server may respond with an error status code. We can handle this using the " onerror " event:

```
xhr.onerror = function(error) {
   console.log(error);
};
```

In the above code, we log the error object to the console if the request fails.

12.4.5 XHR in Practice

12.4.5.1 Making a GET Request

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <div id="result"></div>

    <script src="app.js"></script>
  </body>
```

```
</html>
```

app.js

```
const xhr = new XMLHttpRequest();
const url = "https://jsonplaceholder.typicode.com/posts";

xhr.onload = function () {
   const posts = JSON.parse(xhr.responseText);

   const resultDiv = document.getElementById("result");
   const ul = document.createElement("ul");
   posts.forEach((post) => {
      const li = document.createElement("li");
      li.innerText = post.title;
      ul.appendChild(li);
   });
   resultDiv.appendChild(ul);
};

xhr.onerror = function (error) {
   console.log(error);
};

xhr.open("GET", url);
```

```
xhr.send();
```

The above code uses the XHR API to retrieve JSON data from a server and then display that data on a web page. Here is how it works.

It first creates a new instance of the XMLHttpRequest object, which is used to make HTTP requests to a server. The following line defines the URL from which the JSON data will be retrieved.

An event listener is added to the XMLHttpRequest object's " onload " property, which specifies what should happen once the JSON data has been successfully retrieved.

- In this case, the event listener parses the retrieved JSON data into a JavaScript object using the JSON.parse() method, and then creates a new HTML unordered list element ( ).
- Then, it iterates over each post in the JSON data and creates a new HTML list item ( ), sets the text of the list item to the post's title (post.title), and appends the list item to the unordered list element.
- Finally, the unordered list element is appended to a element.

If an error occurs while making the HTTP request (e.g., the server is not responding), the " onerror " event listener logs the " error " object to the console.

The open() method is called with the HTTP method (GET in this case) and the server URL to which the request will be sent.

The send() method is called to send the HTTP request to the server.

Once the server responds with the JSON data, the “ onload ” event listener is triggered, and the retrieved data is displayed on the page as a list of post titles.

12.4.5.2 Making a POST Request

Here is an example of using the XMLHttpRequest API to send data to a server using the HTTP POST method:

app.js

```
const xhr = new XMLHttpRequest();
const url = "https://jsonplaceholder.typicode.com/posts";

const data = {
  title: "My post title",
  body: "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
  userId: 1,
};

xhr.onload = function () {
  const response = JSON.parse(xhr.responseText);
  console.log(response);
};

xhr.onerror = function (error) {
```

```
    console.log(error);
};

xhr.open("POST", url);
xhr.setRequestHeader("Content-Type", "application/json");

xhr.send(JSON.stringify(data));
```

In this example, we create a new instance of the XMLHttpRequest object and define the URL to which the POST request will be sent.

Next, we define an object called " data " that contains the information we want to send to the server. For example, we send a title, body , and user ID for a new post.

As in the previous example, we then set up event listeners for the " onload " and " onerror " events.

After that, we call the open() method on the XMLHttpRequest object and pass in the HTTP method (POST) and the server URL to which the request will be sent.

We use the setRequestHeader() method to set the " Content-Type " header to " application/json ", which tells the server we are sending JSON data.

Finally, we call the send() method on the XMLHttpRequest object and pass in the " data " object, which we convert to a JSON string using the JSON.stringify() method.

When the server responds, the " onload " event listener is triggered, and we log the response to the console.

12.5 Web Worker API

When JavaScript code runs on a webpage, it can cause the page to become unresponsive until the code finishes executing. Long-running JavaScript code is typically placed in a Web Worker to avoid this issue.

A Web Worker enables JavaScript code to run in the background without disrupting the user's ability to interact with the webpage, such as clicking and selecting items.

12.5.1 Web Worker File

In the below example, we must create a separate JavaScript file for a Web Worker that increases a counter every second. Let's call it "*my-web-worker.js*". Here is the code for the file:

```
var i = 0;

function startCounter() {
   i = i + 1;
   postMessage(i);
   setTimeout("startCounter()", 1000);
}

startCounter();
```

The most critical part of this code is the postMessage() method, which sends a message back to the HTML page.

12.5.2 Web Worker Object

We must set up a Web Worker object in our HTML page to receive messages from a Web Worker. However, we should first check if the web browser supports Web Workers using the following code:

```
if (typeof Worker !== "undefined") {
    // Yes, Web Worker is supported.
    // Your code here...
} else {
    // No, Web Worker is not supported!
}
```

Once we have confirmed Web Worker support, we can create the Web Worker object using the following code:

```
w = new Worker("my-web-worker.js");
```

To receive messages from the Web Worker, we can add an " onmessage " event listener to the Web Worker object:

```
w.onmessage = function (event) {
    document.getElementById("result").innerHTML = event.data;
};
```

Whenever a message is sent by the Web Worker, the code in the event listener will activate, and the data received from the Web Worker will be stored in " event.data ".

To terminate the Web Worker and free up browser resources, we can use the terminate() method and set the variable to undefined :

```
w.terminate();
w = undefined;
```

It is essential to stop the worker when we are finished; otherwise, it will continue to listen for messages (even after the external worker script is done) until it is terminated.

Below is the code for the HTML page:

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <p>Count numbers: <output id="result"></output></p>
    <button onclick="startWorker()">Start Worker</button>
    <button onclick="stopWorker()">Stop Worker</button>

    <script>
      var w;
```

```
        function startWorker() {
            if (typeof Worker !== "undefined") {
                if (typeof w == "undefined") {
                    w = new Worker("my-web-worker.js");
                }
                w.onmessage = function (event) {
                    document.getElementById("result").innerHTML = event.data;
                };
            } else {
                document.getElementById("result").innerHTML = "This browser does not support Web Workers!";
            }
        }

        function stopWorker() {
            w.terminate();
            w = undefined;
        }
    </script>
</body>
</html>
```

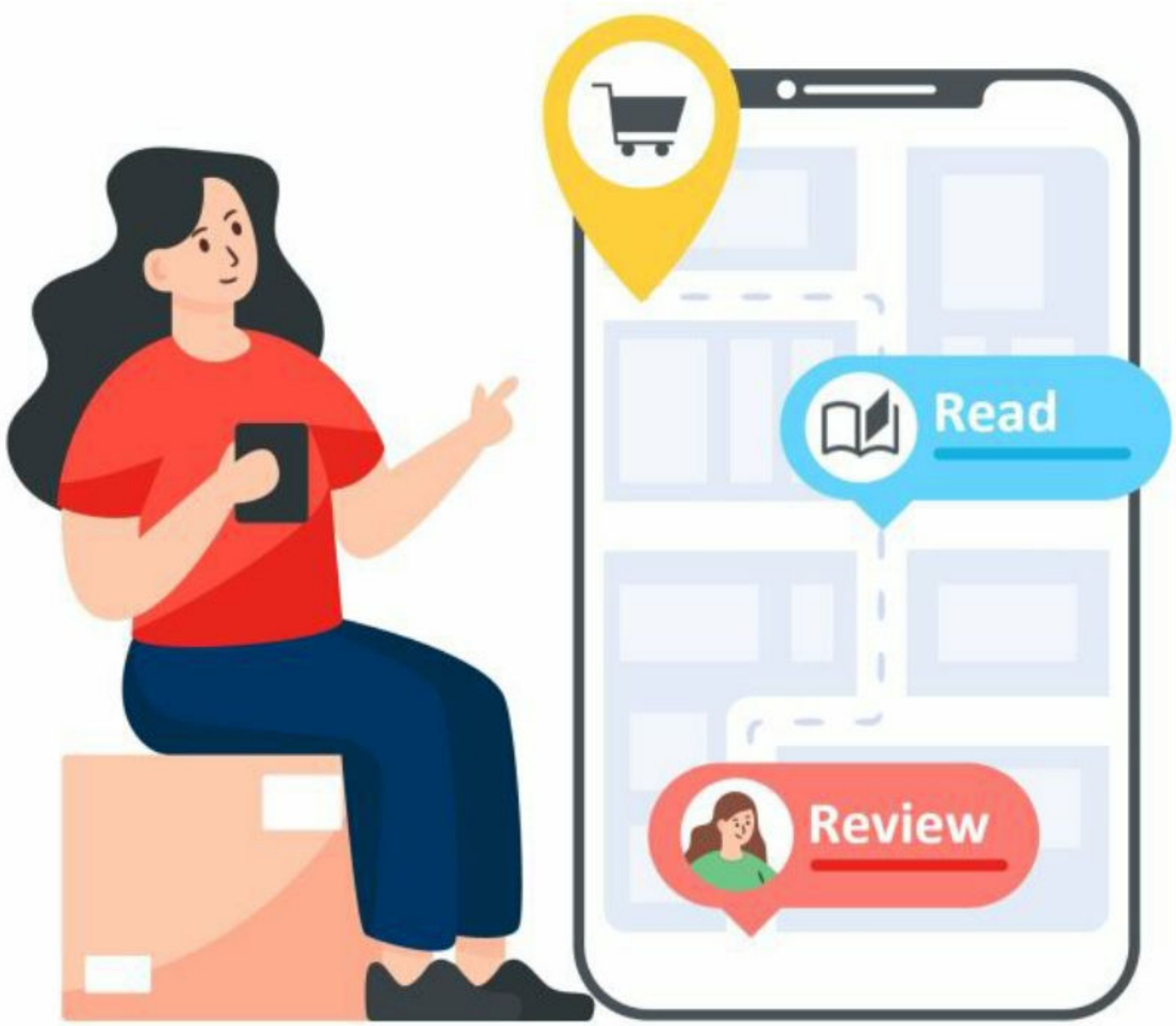
12.5.3 Web Workers and the DOM

Web Workers are placed in external JavaScript files and run in other threads, which means they don't have access to particular JavaScript objects of the Document Object Model (DOM), including:

- The window object
- The document object

In conclusion, Web Workers cannot manipulate or interact with HTML elements on our web pages.

Please Leave a Review on Amazon



Dear Readers,

Your support means the world to me! If you have enjoyed my books, please consider taking a moment to leave a review on Amazon.

Thank you for being part of this journey with me!

Warm regards,

Neo D. Truman

About the Author

Neo D. Truman is a software engineer with over 15 years of experience in full-stack web development. He first developed a passion for technology at a young age and has been working with computers since 1998. In 2011, he obtained a Master of Science in Information Technology degree to further enhance his skills and knowledge in the field.

Neo has gained expertise in software development, web programming, and project management throughout his career. His passion for technology and commitment to excellence is reflected in his work, as he has contributed to the successful completion of numerous high-profile projects.

Neo is also an avid writer and enjoys sharing his knowledge and insights with others. He has authored several technical articles and publications and regularly contributed to various online tech communities. In addition, his ability to explain complex technical concepts in simple terms makes him a valuable resource for novice and experienced developers.

Having a strong desire to assist others, Neo D. Truman is a sought-after expert in the software development community. His dedication to excellence and his commitment to sharing his knowledge make him a valuable asset to anyone seeking to master the art of web development.