HTML : Hyper Text Markup Language

HTML stands for **H**yper**T**ext **M**arkup **L**anguage. It is the standard language used to create and structure the content of a webpage.

- **HyperText:** This refers to the ability to create links that connect web pages to one another, making the web a connected "web" of information.
- **Markup Language:** This means you use "tags" to surround your content, giving that content meaning and structure. You are "marking up" a plain text document.

First Website: https://info.cern.ch/hypertext/WWW/TheProject.html

Install Required Software:

1. Install Visual Studio Code

This is a free piece of software from Microsoft that you will use to write your code.

https://code.visualstudio.com/

2: **Install Visual Studio Code Extensions**

Live Preview

Prettier

Live Server

1: The Heading (<h1> to <h6>)

What it is: Headings are tags used to define titles and subtitles on your page. They are crucial for creating a logical hierarchy and outline for your content.

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

- <h1>: The most important heading, typically used only once per page for the main title.
- <h2>: A major section heading.
- <h3>: A sub-section heading under an <h2>.
- ...and so on, down to <h6>, the least important heading.

The Purpose: To structure your document in a way that is understandable to both humans and machines (like search engines and screen readers). It is **not** just for making text big; it is for giving the text structural importance.

2. The Paragraph (<p>)

What it is: The paragraph tag is the most common tag you'll use. It's for grouping sentences and blocks of text together.

The Tag: <p>

The Purpose: To define a distinct paragraph of text. Browsers automatically add a little bit of space before and after a <p> element, separating it from other content. You should not use multiple line breaks; you should use multiple <p> tags.

```
<p>This is the first paragraph. It contains several sentences about a topic.
</p> <p>This is a second, separate paragraph. The browser will display it
with a space between it and the first one.</p>
```

3: HTML Horizontal Rules

The `<hr>` tag defines a thematic break in an HTML page, and is most often displayed as a horizontal rule.

The `<hr>` element is used to separate content (or define a change) in an HTML page:

```
<p>This is the first topic.</p> <hr /> <p>This is a completely different
topic.</p>
```

4. The Line Break Tag:

This tag tells the browser, "Stop writing on this line and immediately start on the next one." It's like hitting the Enter key once in a poem or an address.

- **Purpose:** Creates a single line break.
- **Example:**codeHtml
- List

```
<p>221B Baker Street<br> London, England</p>
```

6. List Elements (<ul>, <ol>, <li>)

What it is: These tags are used to create lists. This is a perfect example of nesting.

The Tags:

- : An **U**nordered **L**ist. It creates a bulleted list.
- : An **O**rdered **L**ist. It creates a numbered list.
- : A **L**ist **I**tem. Each item in either type of list must be wrapped in an <li> tag.

The Purpose: To group related items together in a list format.

Example (Nesting in action):

The <li> elements are nested as children inside the <ul> parent.

```
<ul> <li>Tea Bag</li> <li>Water</li> <li>Sugar</li> <li>Milk</li> </ul> <ol>
<li>First, boil the water in a kettle.</li> <li>Add tea bag, sugar and milk
into it</li> <li>Keep boiling it for 5 minutes </li> <li>Serve the tea by
pouring it into cup</li> </ol>
```

7: The Anchor Tag (<a>)

What it is: The Anchor tag is what makes the web "hypertext." It is used to create a hyperlink to another webpage, a file, or a location within the same page.

The Tag: <a>

The Purpose: To make text (or an image) clickable, allowing users to navigate. It requires an **attribute** called href (hypertext reference) to specify the destination URL.

```
<a href="https://www.coderarmy.in/">Vist Coder Army</a>
```

HTML Links - The target Attribute

By default, the linked page will be displayed in the current browser window. To change this, you must specify another target for the link.

The `target` attribute specifies where to open the linked document.

The `target` attribute can have one of the following values:

- `_self` - Default. Opens the document in the same window/tab as it was clicked
- `_blank` - Opens the document in a new window or tab

8. The Image Tag (<img>)

What it is: The Image tag is a **self-closing tag** used to embed an image onto your page.

The Tag: <img>

The Purpose: To display a visual image. It requires two main attributes:

- src (source): The path or URL to the image file. This is mandatory.
- alt (alternative text): A description of the image. This is vital for accessibility (screen readers for the visually impaired will read this out) and for when the image fails to load.

```
<img src="https://cdn.britannica.com/16/234216-050-C66F8665/beagle-hound-dog.jpg" height="300px" width="300px" alt="This is Dog Photo">
```

HTML Day03

File Path system

Step 1: The First Principle (The Fundamental Truth)

The most fundamental truth is this: **A website is not one single file.**

It's a collection of different files (HTML, CSS, JavaScript, images, videos, fonts) that all need to work together. These files are organized into folders, just like documents on your computer.

Step 2: The Core Problem

Now we have a core problem: If your index.html file needs to display an image named logo.png, how does the HTML file tell the browser where to **find** logo.png?

You can't just say src="logo.png" and expect it to work every time. What if the logo is in an images folder? What if it's on a completely different website?

The browser needs an exact, unambiguous address to locate the file. **A file path is that address.**

A) Relative File Paths (The Most Important for Your Projects)

A relative file path gives directions to a file **starting from the location of the file you are currently in**. You will use this 99% of the time for linking your own files together (images, CSS, other HTML pages).

```
my-website/ ├── index.html ├── about.html │ ├── images/ │ ├── logo.png │ └── hero.jpg │ └── pages/ ├── contact.html └── terms.html
```

Here are the different scenarios:

Scenario 1: Linking to a file in the SAME folder.

- **The Problem:** You are in index.html and want to link to about.html.
- **The Logic:** They are neighbors, living in the same my-website/ folder. The directions are as simple as possible.
- **The Syntax:** You just use the filename.

```
<!-- This code is inside index.html --> <a href="about.html">About Us</a>
```

Scenario 2: Linking to a file in a SUB-FOLDER (Going Down).

- **The Problem:** You are in index.html and want to display logo.png.
- **The Logic:** From index.html, you need to go *into* the images folder to find the logo.
- **The Syntax:** You write the folder name, a forward slash /, and then the filename.codeHtml

```
<!-- This code is inside index.html --> <img src="images/logo.png" alt="My Website Logo">
```

Scenario 3: Linking to a file in a PARENT FOLDER (Going Up).

- **The Problem:** You are in contact.html (which is inside pages/) and you want to display logo.png (which is inside images/).
- **The Logic:** You can't go directly from pages/ to images/. You must first go *up* one level out of the pages folder to get back to the main my-website/ folder. From there, you can go *down* into the images folder.
- **The Syntax:** Two dots and a slash (../) means "go up one level."codeHtml

```
<!-- This code is inside pages/contact.html --> <img src="../images/logo.p
ng" alt="My Website Logo">
```

  - ../ takes you from pages/ up to my-website/.
  - images/logo.png then takes you down into the images folder to find the file.

B) Absolute File Paths

An absolute file path gives the **full, complete URL** to a resource on the web. It starts with http:// or https://.

When to use it:

You ONLY use absolute paths when you are linking to a resource that is **NOT on your own website**.

- Linking to another website.
- Using a font from Google Fonts.
- Using an image from an image-hosting service.

Example:

codeHtml

```
<!-- Linking to an external website --> <a href="https://www.google.com">Sear
ch on Google</a> <!-- Using an image hosted on another server --> <img src="h
ttps://upload.wikimedia.org/wikipedia/commons/6/6a/Html5_logo_and_wordmark.sv
g" alt="HTML5 Logo">
```

The Critical "Why": Why not use absolute paths for your own files?

A beginner might be tempted to copy the full path from their computer, like this:

src="C:/Users/Arjun/Desktop/my-website/images/logo.png"

This is a major mistake. This address only works on *your* computer. The moment you upload your website to a real web server, that path becomes meaningless and the image will be broken.

Think of a web browser (like Chrome, Firefox, etc.) as a **secure prison** for the code it runs. This prison is called a **"sandbox."**

- **The Inmates:** The HTML, CSS, and JavaScript code you load.
- **The Prison Walls:** The browser itself.

The primary rule of this prison is: **Code running inside the sandbox is NOT allowed to freely access the host computer's file system.**

The Core Problem: A Malicious Website

Imagine if this security rule didn't exist. You could visit a malicious website, evil-website.com, and it could contain the following HTML:

codeHtml

```
<!-- Malicious code --> <img src="C:/Users/Arjun/Documents/MyPrivatePassword
s.txt">
```

If the browser allowed this, the website could potentially read the contents of your private files, your photos, your documents—anything on your computer. It would be a catastrophic security disaster.

Relative paths, however, will always work because they describe the location of files *relative to each other*, no matter what computer or server they are on.

Summary / Rules of Thumb

1. **For Your Own Files (Internal Links): ALWAYS use Relative Paths.**
   - filename.html (Same folder)
   - folder/filename.html (Go down)
   - ../filename.html (Go up)
2. **For Other Websites' Files (External Links): ALWAYS use Absolute Paths.**
   - https://www.example.com/page.html

Absolute Path in Windows

The Windows file system is organized into separate drives, each identified by a letter (like C:, D:, etc.).

- **Starting Point (The Root):** An absolute path in Windows **always starts with a drive letter** followed by a colon and a backslash, like C:\. This is the "root" of that specific drive.
- **Directory Separator:** Windows uses a **backslash ** to separate directories and files in the path.

Windows Structure:

Drive:\Folder\SubFolder\file.txt

Example:

Imagine a file named report.docx located in the Documents folder of a user named JohnDoe on the main C: drive.

The absolute path would be:

C:\Users\JohnDoe\Documents\report.docx

Let's break it down:

- C:\ - Start at the root of the C drive.
- Users - Go into the "Users" folder.
- JohnDoe - Go into the "JohnDoe" folder.
- Documents - Go into the "Documents" folder.
- report.docx - Here is the file.

Absolute Path in macOS (and Linux)

The macOS file system (like Linux and other Unix-like systems) does **not** use drive letters. Instead, it has a single, unified file system.

- **Starting Point (The Root):** An absolute path in macOS **always starts with a single forward slash /**. This symbol represents the one and only root of the entire file system.
- **Directory Separator:** macOS uses a **forward slash /** to separate directories and files. (This is the same separator used in web URLs, which is a helpful thing to remember).

macOS Structure:

/Folder/SubFolder/file.txt

Example:

Let's find the same file: report.docx in the Documents folder for the user johndoe.

The absolute path would be:

/Users/johndoe/Documents/report.docx

Let's break it down:

- / - Start at the absolute root of the entire file system.
- Users - Go into the "Users" directory.
- johndoe - Go into the "johndoe" directory (usernames are typically lowercase on macOS/Linux).
- Documents - Go into the "Documents" directory.
- report.docx - Here is the file.

Summary of Key Differences

Feature	Windows	macOS / Linux
Starting Point (Root)	Drive Letter (e.g., C:\)	A single forward slash (/)
Directory Separator	Backslash (\)	Forward Slash (/)
Example	C:\Users\JohnDoe\Documents\report.docx	/Users/johndoe/Documents/report.docx

HTML Boilerplate Code

Step 1: The First Principle (The Fundamental Truth)

The fundamental truth is that a web browser is a program that needs **specific, predictable instructions** to do its job. It cannot guess your intentions. You can't just give it a file with a <h1> tag and expect it to know what kind of document it is, what language it's in, or how to render it properly.

Step 2: The Core Problem

We need a way to give the browser essential "setup information" *before* it starts rendering our visible content (like paragraphs and images). This setup information needs to answer critical questions:

- **"What version of HTML am I reading?"** (So I know which rules to follow).
- **"What character encoding should I use?"** (So characters like ©, ’, or ₹ are displayed correctly and not as gibberish like â€™).
- **"What should the title of the browser tab be?"**
- **"How should this page behave on a mobile device vs. a desktop?"**

Just writing Hello doesn't answer any of these questions.

Step 3: The Logical Solution - The Boilerplate

The solution is to create a standard, universal starting template—a **boilerplate**. This is a block of code that you will start every single HTML file with. It's not something you need to reinvent each time; you just copy-paste it and fill in the blanks.

It's the "blueprint of the house" that you need before you can start putting in the walls (<h1>) and windows (<img>).

Let's build the modern HTML5 boilerplate from first principles:

The Anatomy of the HTML Boilerplate

codeHtml

```
<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <meta name="vi
ewport" content="width=device-width, initial-scale=1.0"> <title>Document</tit
le> </head> <body> </body> </html>
```

Let's break down each line and explain *why* it's necessary.

1. <!DOCTYPE html>

- **The Problem:** In the past, there were many versions of HTML (HTML4, XHTML, etc.). The browser needed to know which set of rules to use to render the page.
- **The Solution:** This is the very first line and it's called the **Document Type Declaration**. In modern HTML5, it's incredibly simple. This specific line tells the browser: "Use the latest, standard-compliant mode for rendering this HTML document." It's a switch that prevents the browser from falling back into "quirks mode" (an old mode for rendering non-standard pages). **It must always be the first thing in your file.**

2. <html lang="en">...</html>

- **The Problem:** The web is global. How do search engines (like Google) and screen readers (for accessibility) know what human language the content is written in?
- **The Solution:** This is the **root element** that wraps your entire page.
  The lang="en" attribute declares that the primary language of the page content is English. This is very important for accessibility and SEO. You'd change "en" to "es" for Spanish, "hi" for Hindi, etc.

3. <head>...</head>

- **The Problem:** As discussed, we need a place to put all the "behind-the-scenes" information that is for the browser, not for the user to see on the page.
- **The Solution:** The <head> section is the container for this metadata. Nothing you put inside the <head> will be displayed in the main browser window.

4. **<meta charset="UTF-8">**

- **The Problem:** Computers fundamentally only understand numbers. To display text, they use a "character set" to map numbers to letters. There are many different character sets, and if the browser uses the wrong one, your text will be garbled.
- **The Solution:** This meta tag explicitly tells the browser to use **UTF-8**, which is the universal standard character set for the web. It can represent almost any character and symbol from any language in the world. **This line is essential to prevent text-encoding issues.**

5. **<meta name="viewport" ...>**

- **The Problem:** A website can be viewed on a tiny phone screen or a giant desktop monitor. If you don't tell a mobile browser how to handle your page, it will try to render it as if it were on a desktop—resulting in a tiny, zoomed-out, unreadable page.
- **The Solution:** This specific meta tag is the cornerstone of **responsive design**.
  - width=device-width: This tells the browser: "Make the page's width equal to the screen width of the device it's being viewed on."
  - initial-scale=1.0: This sets the initial zoom level to 100% when the page is first loaded.
  - **In simple terms: this line tells the browser to render the page in a way that is optimized for mobile screens.**

6. **<title>...</title>**

- **The Problem:** A user might have 20 tabs open. How do they identify your page? What name should appear when they bookmark your page?
- **The Solution:** The <title> tag sets the text that appears in the browser tab, in bookmark lists, and in search engine results. It's a critical piece of metadata for both usability and SEO.

7. **<body>...</body>**

- **The Problem:** Where do we put the actual visible content?

- **The Solution:** The <body> section is the container for everything the user will actually see on the page: your headings, paragraphs, images, links, tables, etc. All the tags you've learned so far go inside the <body>.

Conclusion: The HTML boilerplate isn't just a random collection of tags. Each line serves a critical, logical purpose to ensure your webpage is rendered correctly, is accessible, is mobile-friendly, and is understood by search engines. It is the non-negotiable starting point for every web page you will ever create.

But website work even without Boilerplate

Browsers are Built to Be Extremely Forgiving.

Think about the early days of the web. It was a chaotic "Wild West." People wrote messy, incorrect, and incomplete HTML all the time. If browsers had been strict—if they had crashed or refused to render a page because of a missing tag—the web would have never taken off.

So, browser makers (like Netscape and Microsoft) made a crucial design decision: **"When in doubt, guess."**

The Core Problem: How to Handle "Bad" HTML?

A browser's job is to display *something*, no matter what you give it. When you give it a file without a boilerplate, here's what happens in the background:

1. **"No <!DOCTYPE html>? Hmm."**: The browser says, "Okay, I don't know what version of HTML this is. To be safe, I'll enter **Quirks Mode**." In quirks mode, the browser tries its best to render the page by mimicking the behavior of very old, non-standard browsers from the late 90s. For a simple <h1>, this usually looks fine, but for more complex layouts (especially with CSS), it can cause strange and unpredictable bugs.
2. **"No <html> or <body> tags? I'll just pretend they're there."**: The browser's parser is smart. It sees content like <h1> that it knows belongs in the <body>. So, it says, "This developer probably forgot the <html>, <head>, and <body> tags. I will **implicitly generate them** in my internal model (the DOM) so I have somewhere to put this <h1>." You don't see these tags in your file, but the browser creates them in its memory to make sense of your document.

3. **"No <meta charset="UTF-8">? I'll guess the encoding."**: The browser will look at the first few bytes of your file and try to guess what character encoding you used. For simple English text, it will almost always guess correctly (e.g., ASCII or Windows-1252). The problem arises when you use special characters (€, —, ’). If the browser guesses wrong, those characters will break.
4. **"No <title>? Fine, I'll use the filename."**: The browser needs something to put in the tab. So, it just uses the name of your HTML file (e.g., index.html) as the default title.

So, Why Use a Boilerplate if the Browser Fixes Everything?

You're right that for a simple "Hello World" page, it *looks* the same. But the boilerplate isn't about making a simple page look right; it's about making a **professional, reliable, and future-proof website.**

Here is why you're playing with fire by omitting it:

1. **Unpredictable Rendering (Quirks Mode):** The biggest danger. Your CSS might work differently across Chrome, Firefox, and Safari because each browser has its own slightly different implementation of quirks mode. What looks good on your machine might look broken on someone else's. Using <!DOCTYPE html> puts all browsers into **Standards Mode**, which is predictable and consistent.
2. **Broken Characters:** Your site might look fine until you need to write "50€" or "resumé". Without charset="UTF-8", those characters can easily break, making your site look unprofessional.
3. **Bad Mobile Experience:** Without the <meta name="viewport"> tag, your website will be almost unusable on a mobile phone. It will appear as a tiny, zoomed-out version of the desktop site.
4. **Poor SEO and Accessibility:** Search engines and screen readers rely on the boilerplate tags (lang, title) to understand your page. A page without them is a "mystery document." It will rank lower in search results and be more difficult for users with disabilities to navigate.

Conclusion:

You can build a shack with a few pieces of wood and no foundation, and it might stand up on a calm day. The boilerplate is the **proper engineering foundation**. It ensures your house (website) will stand up in any weather (any browser), is accessible to everyone (screen readers), is easy to find (SEO), and works on any size of land (any device)

Multipage Website

The fundamental truth is that complex information is almost never presented on a single, infinitely long page. We naturally break information into distinct, self-contained topics. A book has chapters, a store has departments, and a company has different aspects (About, Services, Contact).

A multi-page website is the digital equivalent of this.

The Core Problem

We need a way to create these distinct topic pages and, most importantly, a way for a user to **navigate between them seamlessly**. How do we connect home.html to about.html and contact.html so that they feel like a single, cohesive "website" rather than three disconnected files?

The Logical Solution: A Shared Structure and a Navigation System

The solution involves two key components:

1. **A Consistent Structure:** All pages on the site should share a common look and feel. They should have the same header (with the logo and navigation) and the same footer. This reassures the user that they are still on the same website.
2. **A Linking System:** We need to use the <a> (anchor) tag to create a navigation menu that appears on *every single page*, providing reliable doorways to all other pages.

1. The (The Generic Box)

First Principle (The Fundamental Truth)

We need a way to group different elements (<h1>, <p>, <img>, etc.) together into a single, conceptual unit.

The Core Problem

Imagine you have a "user profile card" on a webpage. It consists of an image, a heading for the user's name, and a paragraph for their bio.

codeHtml

```
<img src="avatar.png"> <h2>Arjun Kumar</h2> <p>Loves to code and teach HTML.
</p>
```

These are three separate elements. What if you want to put a border around *all three of them* as a single block? Or give them all a shared background color? Or move them all to the right side of the page as one unit? You have no way to target them as a group.

The Logical Solution: The Generic Container

The most basic solution is to invent a generic, **meaningless** container. Its only job is to be a box that you can put other things into. It shouldn't have any inherent meaning or style; it's just a grouping mechanism.

This is the ** ** (short for "division").

By wrapping our elements in a , we create a single "box" that we can now control.

codeHtml

```
<div> <img src="avatar.png"> <h2>Arjun Kumar</h2> <p>Loves to code and teach
HTML.</p> </div>
```

Now, using CSS, we can say "put a border on that " and it will wrap around the entire group.

2. class and id (The Labels for the Box)

First Principle

Once we have boxes (s), we need a way to identify and find them.

The Core Problem

Your webpage might have ten different profile cards, a sidebar, a main content area, and a photo gallery—all made of s. How do you tell the browser (specifically, your CSS or JavaScript) which box you want to style? How do you say, "Make *all* the profile cards have a gray background, but make the *one* featured profile card have a gold border?" You need a targeting system.

The Logical Solution: Two Types of Labels

The logical solution is to invent two types of labels (which we call **attributes**) that you can add to any HTML tag.

1. **A Unique Identifier:** We need a label for one, and only one, specific element on the entire page. It must be unique. This is perfect for major, one-of-a-kind layout sections like the main navigation bar or a search form. This is the **id**.
2. **A Reusable Classifier:** We also need a label that we can apply to *multiple* elements to group them into a category. This is for things that have a similar style or function, like all the profile cards, all the error messages, or all the "buy now" buttons. This is the **class**.

Analogy:

- An **id** is like a person's unique **Social Security Number or Aadhaar Number**. Only one person can have it.
- A **class** is like a person's **Job Title** (e.g., "Engineer"). Many people in a company can have the class "Engineer".

Example:

codeHtml

```
<!-- The one and only main header on the page --> <div id="main-header">...</
div> <!-- A card that is special --> <div id="featured-profile" class="profil
e-card">...</div> <!-- A standard card --> <div class="profile-card">...</div
> <!-- Another standard card --> <div class="profile-card">...</div>
```

Now in CSS, you can target them:

- #main-header { ... } (Targets the one unique element with that ID)
- .profile-card { ... } (Targets ALL THREE elements with that class)

- #featured-profile { ... } (Targets only the one special card to give it a gold border)

3. The (The Inline Highlighter)

First Principle

Sometimes we need to label and style a piece of content *within* a line of text, without creating a new block.

The Core Problem

What if you want to make just a *single word* inside a paragraph red?

This is very important information.

You need to select the word "important". If you wrap it in a , it will create a line break, ruining the sentence:

This is very important information. --> Renders incorrectly.

The Logical Solution: The Generic Inline Container

We need a container that works just like a , but is **inline**, meaning it does *not* create a line break. It just wraps a small piece of content within the natural flow of the text.

This is the ****.

codeHtml

```
<p>This is very <span class="highlight">important</span> information.</p>
```

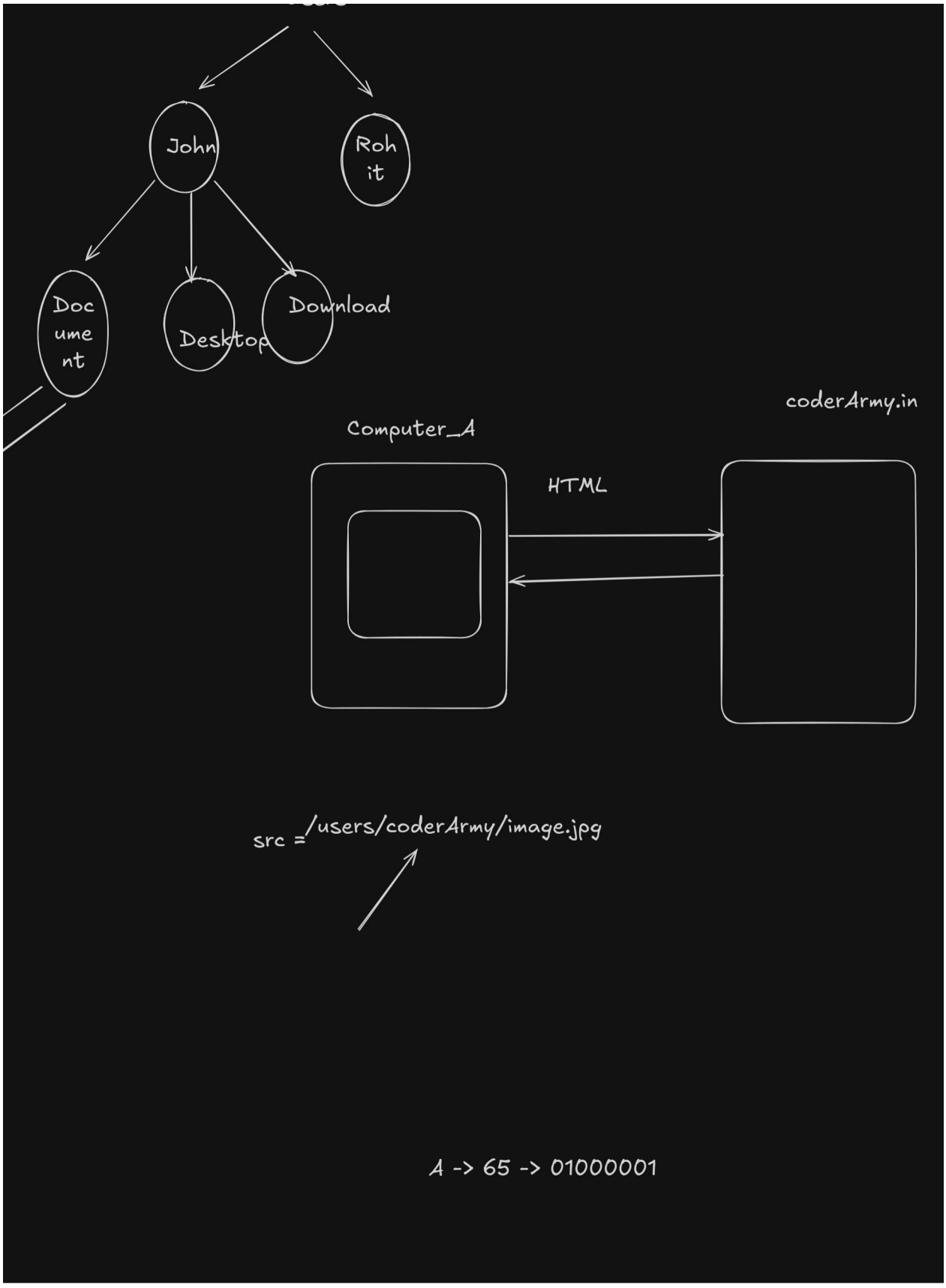
Now you can use CSS to target the .highlight class and make that one word red, without affecting the layout of the paragraph.

Analogy: A is like using a **highlighter pen**. You can mark a few words in a book without having to rip out the page and put it in a separate box.

- <**header**>: This is the box for the introductory content at the top of your page or a section. It typically contains your logo, site navigation (<nav>), and main heading.
- <**footer**>: The box for the closing content at the bottom. It usually contains copyright info, contact details, and secondary links.
- <**main**>: This is the most important one. It defines the **main, unique content** of that specific page. It should not contain things that are repeated on every page (like the header or footer). There should only be **one** <main> tag per page.

notes:


root
lion.png
cat.png
Alpha
dog.png
index.html
Beta
home.html
cow.png
Photos
First.png
second.png
../cat.png
File Path
<img src = "./dog.png">
dow.png
Relative Path:
Absolute Path:
./Beta/cow.png
Beta/Photos/secon
../../cat.png
Absoulte Path:
Windows: C:\Users\JohnDoe\Documents\report.docx
Linux(mac): /Users/Johndoe/Documents/report.docx
/Users/coderarmy/Desktop/puppy.jpeg
root
C
Users
Admin


John
Roh
it
Doc
ume
nt
Desktop
Download
coderArmy.in
Computer_A
HTML
src = /users/coderArmy/image.jpg
A -> 65 -> 01000001

Forms in HTML

The First Principle: The Web Needs to Be a Two-Way Street

The fundamental truth is that a website isn't just a brochure for you to read. For the web to be useful, it needs a way to **collect information *from* the user** and send it back to the server.

Without this, you couldn't log in, search for a video, buy a product, post a comment, or send a message. The web would be a read-only library.

The Core Problem

How do we create a standardized, reliable system to:

1. **Display** interactive fields for a user to fill in (text boxes, checkboxes, dropdowns).
2. **Package** the user's data neatly.
3. **Send** that package to a specific destination on a server.
4. **Tell** the server *how* the data is being sent.

The logical solution to this entire problem is the HTML **<form>**.

The <input> Element: The Box for Your Stuff

This is the most common form tag. It's a self-closing tag that creates an input field. Its behavior changes based on its type attribute.

The most basic type is text.

```
<input type="text">
```

```
<form> <input type="text"> </form>
```

The Need for Meaning - The <label>

The Problem: We have a box, but the user has no idea what they are supposed to type into it. Is it for a name? An email? A search query? The box is meaningless without a description.

The Solution: We need to add a descriptive piece of text. The correct HTML tag for this is the <label>. It's a tag specifically designed to be the title for a form field.

Let's add a label:

codeHtml

```
<label>First Name:</label> <input type="text">
```

Result: This is better! Now the user sees "First Name:" next to the box and knows what to type. But the <label> and the <input> are still two completely separate, unrelated things. The browser doesn't know they belong together

The Need for Connection - id and for

The Problem: How can we create a direct, unbreakable link between the label "First Name:" and its specific input box? We need this for two reasons:

1. **Usability:** It would be great if a user could click on the *text* of the label to activate the input box.
2. **Accessibility:** Screen readers for visually impaired users need to know which label describes which input so they can announce it correctly.

The Solution: We need a unique naming system.

1. First, we give our input box a unique name that no other element on the page has. The attribute for a unique name is **id**. Let's give it an id of "firstName".
2. Next, we tell the label which element it is **for**. The for attribute on the label must match the id of the input.

Let's connect them:

```
<label for="firstName">First Name:</label> <input type="text" id="firstName">
```

Result: We have now created a powerful, explicit connection.

- **Try it:** If you click on the text "First Name:", your cursor will magically jump into the text box.
- **Behind the scenes:** A screen reader will now announce, "First Name, edit text" when the user focuses on the input box. The two elements are now a true pair.

The Need for Submission - The <form> and submit

The Problem: We have a field for the user to fill out, but we have no way for them to actually *submit* this information. We need a container for our fields and a "Go" button.

The Solution:

1. We wrap all our form fields in a <form> tag. This tag acts as the main container that tells the browser, "Everything inside here is part of one single submission."
2. We add a button that tells the form to submit. The simplest way is <input type="submit">.

Let's build the form structure:

codeHtml

```
<form> <label for="firstName">First Name:</label> <input type="text" id="firs
tName"> <br><br> <!-- We'll use simple line breaks for spacing for now --> <l
abel for="lastName">Last Name:</label> <input type="text" id="lastName"> <br>
<br> <input type="submit"> </form>
```

Result: We now have a complete visual form with two fields and a submit button. When you click the button, the page reloads, but the data doesn't go anywhere yet.

The Need for Data Identification - The name Attribute

The Problem: When the form is submitted, the browser needs to package the data to send to a server. How does it label the data? If a user types "Arjun" in the first box, how does the server know that "Arjun" is the firstName? The id attribute is only for use *within* the page; it is not sent to the server.

The Solution: We need another attribute whose sole purpose is to be the "**data label**" or the "**key**" for the submitted value. This is the **name** attribute.

Let's add names to our inputs:

```
<form> <label for="firstName">First Name:</label> <input type="text" id="firs
tName" name="firstName"> <br><br> <label for="lastName">Last Name:</label> <i
nput type="text" id="lastName" name="lastName"> <br><br> <input type="submi
t"> </form>
```

Result: Now we have a truly functional form, ready to send meaningful data. When submitted, the browser will create a package that looks like this:

- firstName = (whatever the user typed)
- lastName = (whatever the user typed)

The Need for "Select One" - Radio Buttons

The Problem: What if you want to ask a question where the user can only choose **one option** from a predefined list? For example, "What is your gender?" or "What is your T-shirt size (Small, Medium, Large)?" A text box is a bad solution—users could type anything ("Med", "M", "medium"), making the data inconsistent.

The Solution: We need an input type where selecting one option automatically de-selects all others. This is the **radio button**: <input type="radio">.

This introduces a new rule. How does the browser know which radio buttons belong to the same question?

The Rule: All radio buttons in a single group **must share the same name attribute**. The name acts as the group identifier.

Let's build a T-shirt size selector:

```
<!-- We'll add this inside our existing <form> --> <label>T-Shirt Size:</label> <br> <!-- All three are part of the "shirtSize" group --> <input type="radio" id="sizeS" name="shirtSize" value="small"> <label for="sizeS">Small</label> <br> <input type="radio" id="sizeM" name="shirtSize" value="medium"> <label for="sizeM">Medium</label> <br> <input type="radio" id="sizeL" name="shirtSize" value="large"> <label for="sizeL">Large</label> <br><br>
```

Let's break down the new attributes:

- type="radio": Creates the circular radio button.
- name="shirtSize": This is the **critical** part. Because all three have the same name, the browser knows they are a single group and will only let you select one.
- id="sizeS": Each input still needs a unique id so its specific label can connect to it.
- value="small": This is the actual data that will be sent to the server if this option is selected. If the user clicks "Small", the form will send shirtSize=small. Without the value, the data would be meaningless.

The Need for "Select Many" - Checkboxes

The Problem: Now, what if you want to ask a question where the user can choose **multiple options**? For example, "Which toppings would you like on your pizza?" A radio button won't work, because you can only select one.

The Solution: We need an input type that allows for multiple selections. This is the **checkbox**: <input type="checkbox">.

Checkboxes that are part of the same question should also share the same name. This tells the server that all the selected values belong to the same category ("toppings").

Let's build a toppings selector:

codeHtml

```
<label>Pizza Toppings:</label> <br> <input type="checkbox" id="toppingPep" name="toppings" value="pepperoni"> <label for="toppingPep">Pepperoni</label> <br> <input type="checkbox" id="toppingMush" name="toppings" value="mushrooms"> <label for="toppingMush">Mushrooms</label> <br> <input type="checkbox" id="toppingOni" name="toppings" value="onions"> <label for="toppingOni">Onions</label> <br><br>
```

Breakdown:

- type="checkbox": Creates the square checkbox.
- name="toppings": All three share this name, telling the server they are all "toppings".
- id="toppingPep": Each has a unique id for its label.
- value="pepperoni": Each has a unique value to identify which topping was chosen.

Result: You now have three checkboxes, and you can click and select as many as you want.

A Better Button - The <button> Element

The Problem: Our <input type="submit"> works, but it's very limited. You can only put plain text in it using the value attribute. What if you want a button with an image, or with bold text?

The Solution: Use the <button> element. It's a container tag, meaning it has an opening and closing tag. This allows you to put other HTML elements *inside* it.

Let's replace our old submit button:

codeHtml

```
<!-- OLD WAY --> <input type="submit" value="Submit Your Order"> <!-- NEW, BE
TTER WAY --> <button type="submit"> <strong>Submit</strong> Your Order </butt
on>
```

Breakdown:

- <button>: The container for the button.
- type="submit": This is very important. This attribute tells the button to act as a form submit button. (It can also be type="button" for JavaScript or type="reset").
- Submit: We can now put other HTML tags, like or even an <img>, right inside our button!

Result: A more flexible and powerful button that has the exact same submit functionality. From now on, we'll prefer <button type="submit">.

The Need for Long-Form Text - <textarea>

The Problem: Our <input type="text"> is great for single lines of text like a name, but it's terrible for longer input, like a user comment or a shipping address. The text just scrolls sideways and becomes unreadable.

The Solution: We need a dedicated element for multi-line text input. This is the **<textarea>** tag.

Unlike <input>, <textarea> is a container tag (it has an opening and closing tag). It's also linked to a <label> using the same for and id pattern.

Let's add a comments box:

codeHtml

```
<label for="comments">Any special instructions?</label> <br> <textarea id="co
mments" name="comments" rows="4" cols="50"></textarea> <br><br>
```

- rows="4": This attribute controls the visible height of the text area, suggesting it should be about 4 lines of text tall.
- cols="50": This controls the visible width, suggesting it should be about 50 characters wide.

The Need for Many Options - The Dropdown (<select>)

The Problem: Radio buttons are good for 3-4 options, but what if you need the user to select one option from a very long list, like their country? A list of 200 radio buttons would make the page incredibly long and difficult to use.

The Solution: A dropdown menu. It compactly hides all the options until the user clicks on it. This is created with the **<select>** tag, which contains multiple **<option>** tags.

Let's build a country selector:

codeHtml

```
<label for="country">Country:</label> <br> <select id="country" name="countr
y"> <option value="">--Please choose an option--</option> <option value="in">
India</option> <option value="us">USA</option> <option value="uk">United King
dom</option> <option value="au">Australia</option> </select> <br><br>
```

Breakdown:

- **<select>**: This is the main container for the dropdown. The id and name attributes go on this tag.
- **<option>**: Each individual choice in the dropdown is an <option> tag.
- **value Attribute on <option>**: This is the data that gets sent to the server. The text between the tags (India) is what the user sees.

More Specialized Inputs (HTML5 Power-ups)

HTML5 introduced many new type attributes for <input> to make forms smarter and more user-friendly.

type="password"

The Problem: We need a text field for sensitive information that shouldn't be visible on the screen as the user types.

The Solution: <input type="password">. It masks the input with dots or asterisks.

```
<label for="userPass">Password:</label> <br> <input type="password" id="userPass" name="userPassword"> <br><br>
```

type="number" with min and max

The Problem: We want the user to enter a number, like their age, but we want to restrict the input to a valid range. We also want mobile browsers to show a number keypad.

The Solution: <input type="number"> with min and max attributes for validation.

```
<label for="userAge">Age (18-99):</label> <br> <input type="number" id="userAge" name="age" min="18" max="99"> <br><br>
```

Result: This creates a number field, often with small up/down arrows. The browser will prevent the form from submitting if the user enters a number outside the 18-99 range.

type="date"

The Problem: Asking users to type a date in a specific format (e.g., MM/DD/YYYY) is prone to errors.

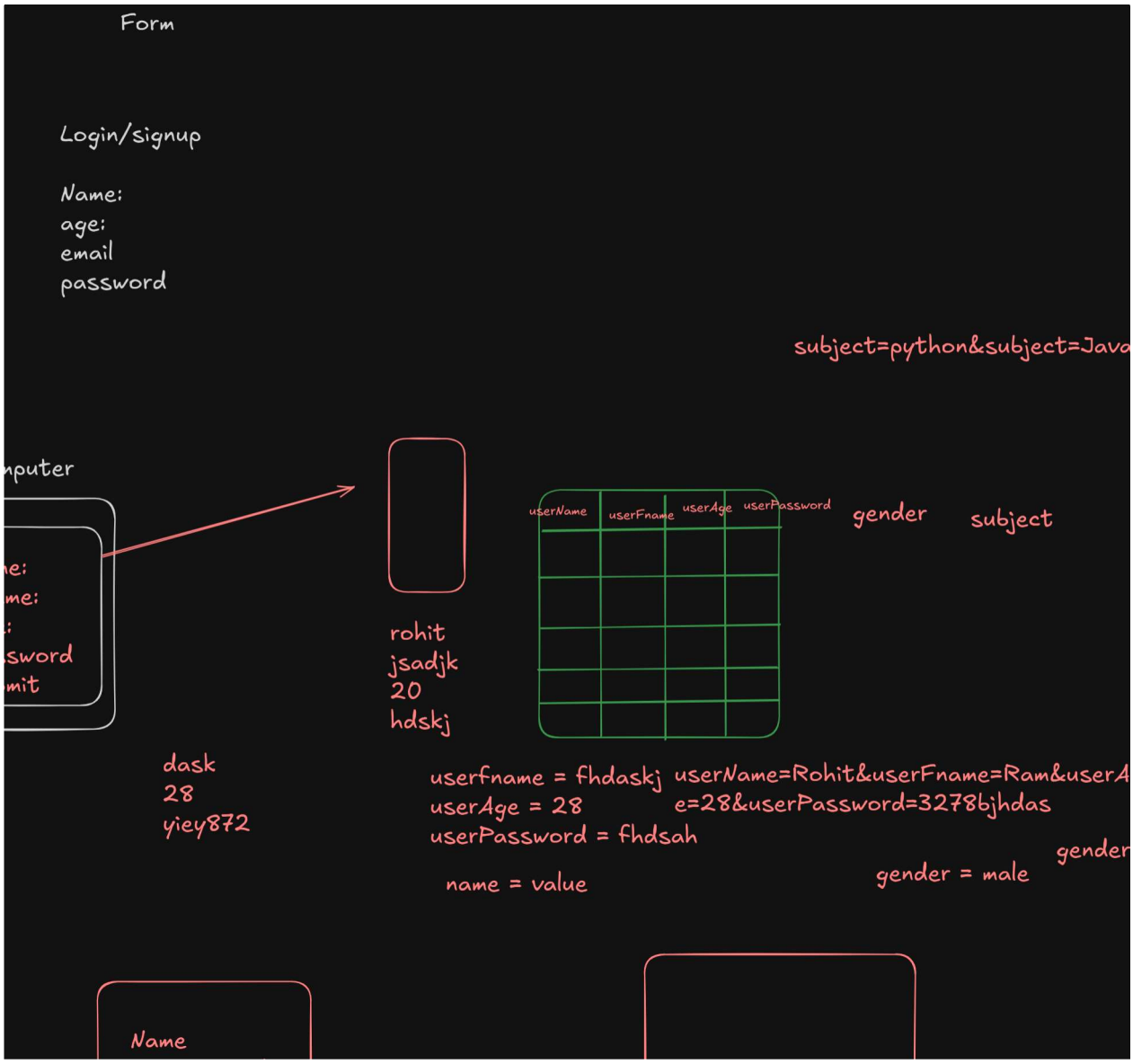
The Solution: <input type="date">. Most browsers will display a user-friendly calendar date picker.

codeHtml

```
<label for="birthDate">Date of Birth:</label> <br> <input type="date" id="bir
thDate" name="dob"> <br><br>
```

Other Useful Types for Homework

- **type="color"**: Displays a color picker.
- **type="range"**: Creates a slider control.
- **type="file"**: Allows the user to upload a file from their device.
- Required and placeholder


Form
Login/signup
Name:
age:
email
password
subject=python&subject=Java
nputer
ne:
me:
sword
mit
rohit
jsadjk
20
hdskj
userName
userFname
userAge
userPassword
gender
subject
dask
28
yiey872
userfname = fhdaskj
userAge = 28
userPassword = fhdsah
userName=Rohit&userFname=Ram&userA
e=28&userPassword=3278bjhdas
name = value
gender = male
gender
Name

HTML MEDIA

First Principle: The Web is More Than Text and Images

The fundamental truth is that a webpage should be able to deliver any kind of content, not just static text and pictures. For years, this was a major problem. To play a video or audio file, browsers had to rely on third-party plugins like Adobe Flash, QuickTime, or Silverlight. This was inefficient, insecure, and inconsistent across different computers.

The Core Problem

How can we embed video and audio into a webpage in a **standardized, native, and plugin-free** way? We need a universal system that works on every modern browser, from a desktop PC to a mobile phone, without requiring the user to install anything extra.

The Logical Solution: The HTML5 <video> and <audio> Tags

The solution was to create dedicated HTML tags whose sole purpose is to embed and control media.

Part 1: The Basics - Getting a Video on the Page

The star of the show is the <video> tag. At its simplest, it just needs to know the source of the video file.

code Html

downloadcontent_copy

expand_less

```
<!-- This is the most basic implementation --> <video src="my-awesome-video.m
p4"></video>
```

The Problem We Immediately Face: If you put this on a page, you'll just see the first frame of the video as a static image. You can't play it, pause it, or change the volume. It's not a video player; it's just a video frame.

The Solution: We need to add player controls. The browser has a beautiful set of default controls built-in, and we can turn them on with a simple attribute.

The controls Attribute: The Magic Switch

The controls attribute is a **boolean attribute**. You don't need to set it to a value (controls="true" is unnecessary); its mere presence turns the feature on.

code Html

downloadcontent_copy

expand_less

IGNORE_WHEN_COPYING_START

IGNORE_WHEN_COPYING_END

```
<video src="my-awesome-video.mp4" controls></video>
```

Result: You now have a fully functional video player on your page, complete with a play/pause button, a timeline scrubber, volume controls, and a fullscreen option. This is the simplest, most effective way to embed a video.

Part 2: Intermediate - Gaining More Control with Attributes

Now that we have a player, how do we control its behavior and appearance? We use attributes.

- **width and height**: Just like with an <img> tag, you can set the dimensions of the player. code Html

  downloadcontent_copy

  expand_less

  IGNORE_WHEN_COPYING_START

  IGNORE_WHEN_COPYING_END

  ```
  <video src="video.mp4" width="640" height="360" controls></video>
  ```

- **autoplay**: This attribute will make the video start playing as soon as the page loads. code Html

  CRITICAL CAVEAT: Modern browsers (Chrome, Safari, Firefox) will **block autoplay with sound** because it's a terrible user experience. To make autoplay work, you almost always have to add the muted attribute as well.

  downloadcontent_copy

  expand_less

  IGNORE_WHEN_COPYING_START

  IGNORE_WHEN_COPYING_END

  ```
  <!-- This will autoplay, but with no sound --> <video src="video.mp4" auto
  play muted controls></video>
  ```

- **loop**: Makes the video automatically restart from the beginning when it finishes. Great for background videos.
- **poster**: This is a fantastic feature for user experience. It specifies an image to display before the video is played, just like a YouTube thumbnail. code Html

  downloadcontent_copy

  expand_less

  IGNORE_WHEN_COPYING_START

  IGNORE_WHEN_COPYING_END

  ```
  <video src="video.mp4" poster="images/video-thumbnail.jpg" controls></vide
  o>
  ```

Part 3: The <audio> Tag

The good news is that the <audio> tag works almost identically to the <video> tag, just without the visual component.

The Problem: How do we embed a sound file (like a song or a podcast) with player controls?

The Solution: Use the <audio> tag with the controls attribute.

code Html

downloadcontent_copy

expand_less

IGNORE_WHEN_COPYING_START

IGNORE_WHEN_COPYING_END

```
<p>Listen to our theme song:</p> <audio src="audio/theme-song.mp3" controls>
</audio>
```

Result: A clean audio player with play/pause, a timeline, and volume controls. It uses the same attributes like autoplay, loop, and muted.

Part 4: Advanced & Professional - Solving Real-World Problems

This is where we move from just making a player work to making it work for *everyone*, on *every browser*.

Problem #1: Not all browsers support the same video formats.

Chrome might prefer the modern .webm format, while Safari on an iPhone might only support .mp4. If you only provide one src, some of your users won't be able to see your video.

The Solution: The <source> Element

Instead of putting the src on the <video> tag itself, you can provide multiple formats *inside* the tag using the <source> element. The browser will go down the list and play the **first one it supports**.

code Html

downloadcontent_copy

expand_less

IGNORE_WHEN_COPYING_START

IGNORE_WHEN_COPYING_END

```
<video controls poster="images/thumbnail.jpg"> <source src="videos/my-video.webm" type="video/webm"> <source src="videos/my-video.mp4" type="video/mp4"> Sorry, your browser doesn't support embedded videos. </video>
```

This is the robust, professional way to embed media. The type attribute tells the browser what kind of file it is so it doesn't have to waste time downloading a file it can't play.

Problem #2: Your video is not accessible.

Users who are deaf or hard of hearing can't understand your video. Users in a noisy environment (or a quiet office) can't listen to the audio. Users speaking another language won't understand it.

The Solution: The <track> Element

The <track> element allows you to add timed text tracks, such as subtitles or captions. It's a self-closing tag that points to a special text file, usually in **WebVTT (.vtt) format**.

code Html

downloadcontent_copy

expand_less

IGNORE_WHEN_COPYING_START

IGNORE_WHEN_COPYING_END

```
<video controls> <source src="video.webm" type="video/webm"> <source src="video.mp4" type="video/mp4"> <!-- English Captions for the hearing impaired --> <track src="captions_en.vtt" kind="captions" srclang="en" label="English"> <!-- Spanish Subtitles for translation --> <track src="subtitles_es.vtt" kind="subtitles" srclang="es" label="Español"> </video>
```

Breakdown:

- src: Path to the .vtt file.

- kind: The type of track. The most common are:
  - captions: A direct transcription of the dialogue and important sounds, for users who can't hear the audio.
  - subtitles: A translation of the dialogue into another language.
  - descriptions: A separate audio description for visually impaired users.
- srclang: The language of the track file (e.g., "en" for English).
- label: The name that appears in the video player's captions menu.

This makes your video accessible to a much wider audience and is a critical part of professional web development.

The Robust & Professional Method (<source>)

codeHtml

```
<video width="320" height="240" controls> <source src="movie.mp4" type="vide
o/mp4"> <source src="movie.ogg" type="video/ogg"> Your browser does not suppo
rt the video tag. </video>
```

This method's primary purpose is **to ensure maximum browser compatibility.**

The First Principle: Not all web browsers support the same video file formats.

- **.mp4 (H.264 codec):** This is the most widely supported format today. Almost all modern browsers can play it.
- **.webm (VP8/VP9 codec):** This is an open-source format heavily promoted by Google. It has excellent support in Chrome and Firefox.
- **.ogg (Theora codec):** This was an older open-source alternative, more popular before .webm took over.

How It Works (The Logic):

This code gives the browser a **list of options**. The browser will read this list from top to bottom and play the **very first video format it understands**.

1. The browser first sees <source src="movie.mp4" ...>. It asks itself, "Can I play MP4 video?"
    - If the answer is **YES**, it loads movie.mp4, plays it, and **completely ignores the rest of the <source> tags**.
    - If the answer is **NO**, it moves to the next option.
2. The browser then sees <source src="movie.ogg" ...>. It asks, "Okay, can I play Ogg video?"
    - If the answer is **YES**, it loads movie.ogg and plays it.
    - If the answer is **NO**, it has no other options.

The Fallback Text:

The text "Your browser does not support the video tag." is the final fallback. It will **only** be displayed if the user is on an extremely old browser (like Internet Explorer 8) that doesn't even recognize what the <video> tag is.

In Summary (Example 1):

- **Pros:** The most reliable and professional way to embed video. It provides multiple formats to ensure your video will play for the largest possible audience across different browsers and devices.
- **Cons:** Requires you to have your video file encoded in multiple formats, which takes extra work.

Example 2: The Simple & Direct Method (src attribute)

codeHtml

```
<video src="a.mp4" controls></video>
```

This method's primary purpose is **simplicity and directness.**

How It Works (The Logic):

This code gives the browser a **single, direct command**: "Play the file a.mp4." There are no other options.

- The browser sees the src attribute and asks itself, "Can I play MP4 video?"
  - If the answer is **YES**, it loads a.mp4 and plays it.
  - If the answer is **NO**, the video simply will not play. The user will likely see an error message or a black box.

In Summary (Example 2):

- **Pros:** Very simple to write and easy to read. You only need one video file.
- **Cons:** It's an "all or nothing" approach. If the user's browser doesn't support the .mp4 format for some reason, the video fails completely. There is no fallback option.

Comparison Table

Feature	Example 1 (<source> Method)	Example 2 (src Method)
Browser Compatibility	**Excellent.** Provides multiple formats as fallbacks.	**Good, but not guaranteed.** Relies on a single format.
Flexibility	High. You can list many different formats.	Low. Only one format is possible.
Simplicity	More complex. Requires multiple lines and files.	Very simple. Requires only one line.
Fallback for Old Browsers	Excellent. Provides custom text if the <video> tag fails.	None.
Best Use Case	**Public-facing websites** where you need to support all users.	**Quick tests, internal projects,** or when you are 100% certain your target audience supports your one format.

Conclusion: Which One Should You Use?

For any serious, public website, **you should always prefer the first method using the <source> element**. It is the industry best practice.

However, since .mp4 has become so universally supported in the last few years, many developers now take the shortcut and use the second, simpler method for convenience, especially for internal or less critical projects. But they are accepting the small risk that it might not work for every single user.

The Two Ways to Create the Copyright Symbol ©

You can use either the **entity name** or the **entity number**. Both produce the exact same result. The entity name is generally easier to remember.

1. Using the Entity Name (Recommended)

This is the most common and readable method.

Code: ©

Example Usage in a Footer:

codeHtml

```
<footer> <p>Copyright &copy; 2024 My Awesome Website. All Rights Reserved.</p
> </footer>
```

How it renders in the browser:

Copyright © 2024 My Awesome Website. All Rights Reserved.

---

2. Using the Entity Number

This method uses the character's numerical code. It works just as well but is less descriptive.

Code: ©

Example Usage:

codeHtml

```
<footer> <p>Copyright &#169; 2024 My Awesome Website. All Rights Reserved.</p
> </footer>
```

How it renders in the browser:

Copyright © 2024 My Awesome Website. All Rights Reserved.

Other Common Symbol Entities You Should Know

The same principle applies to other common symbols.

Symbol	Description	Entity Name	Entity Number
©	Copyright	©	©
®	Registered Trademark	®	®
™	Trademark	™	™
<	Less-than (for showing code)	<	<
>	Greater-than (for showing code)	>	>
&	Ampersand	&	&
"	Double Quote	"	"