CSS

Chapter 1: About CSS and Syntax

Reviewing CSS syntax

CSS Syntax:

CSS (Cascading Style Sheets) is used to style HTML elements on a webpage.

To apply styles, you use a selector to target the HTML element you want to style.

CSS Selector:

The selector is like a pointer that indicates which HTML element you want to style.

For example, if you want to style all paragraphs, you use the selector p.

Declaration Block:

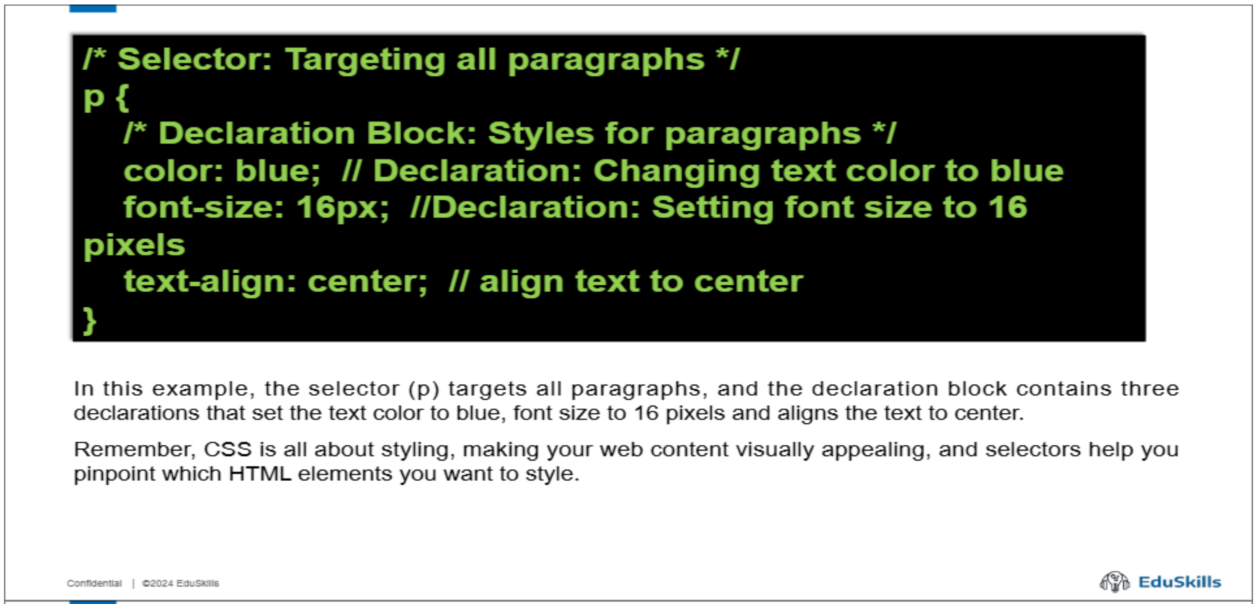
The declaration block is where you define the styles for the selected HTML element.

It is enclosed in curly braces { }.

Declarations:

Inside the declaration block, you have one or more declarations separated by semicolons ;.

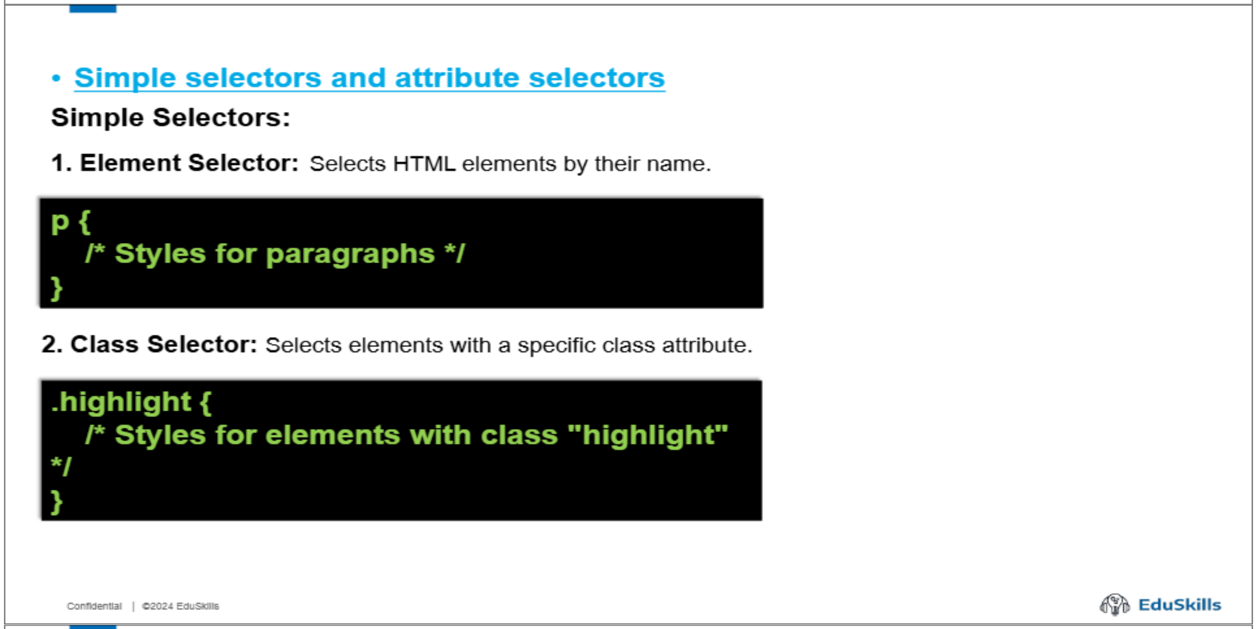
Each declaration consists of a CSS property name and a value, separated by a colon :.



```
/* Selector: Targeting all paragraphs */
p {
   /* Declaration Block: Styles for paragraphs */
   color: blue;  // Declaration: Changing text color to blue
   font-size: 16px;  //Declaration: Setting font size to 16
pixels
   text-align: center;  // align text to center
}
```

In this example, the selector (p) targets all paragraphs, and the declaration block contains three declarations that set the text color to blue, font size to 16 pixels and aligns the text to center.

Remember, CSS is all about styling, making your web content visually appealing, and selectors help you pinpoint which HTML elements you want to style.



- **Simple selectors and attribute selectors**

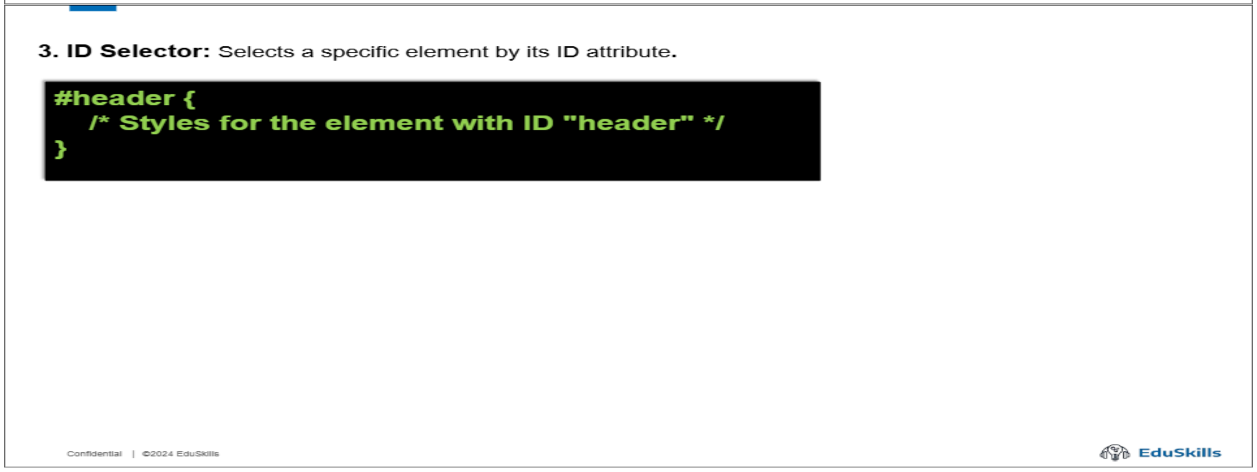
Simple Selectors:

1. Element Selector: Selects HTML elements by their name.

```
p {
   /* Styles for paragraphs */
}
```

2. Class Selector: Selects elements with a specific class attribute.

```
.highlight {
   /* Styles for elements with class "highlight"
*/
}
```



3. ID Selector: Selects a specific element by its ID attribute.

```
#header {
   /* Styles for the element with ID "header" */
}
```

Attribute Selectors:

1. Attribute Exists Selector: Selects elements with a specified attribute.

```
[target] {
   /* Styles for elements with a "target" attribute */
}
```

2. Attribute Value Selector: Selects elements with a specific attribute value.

```
[type="text"] {
   /* Styles for elements with type attribute set to
"text" */
}
```

3. Attribute Starts With Selector: Selects elements with an attribute that starts with a specific value.

```
[class^="btn"] {
   /* Styles for elements with class starting with "btn" */
}
```

4. Attribute Ends With Selector: Selects elements with an attribute that ends with a specific value.

```
[href$=".pdf"] {
   /* Styles for elements with href ending in ".pdf" */
}
```

5. Attribute Contains Selector: Selects elements with an attribute that contains a specific value.

```
[src*="image"] {
   /* Styles for elements with src containing "image" */
}
```

NOTE: Selectors play a crucial role in targeting specific elements on a webpage, and understanding simple selectors and attribute selectors gives you the power to apply styles precisely to achieve the desired visual presentation.

Pseudo-Class Selectors:

Pseudo-class selectors are used to select and style elements based on their state or position in the document structure. They are denoted by a colon (:) followed by the pseudo-class name. Here are some commonly used pseudo-class selectors:

1.: **hover-** Selects and styles an element when the mouse is over it.

```
button:hover {
  background-color: #4CAF50;
  color: white;
}
```

2. **:active**: Selects and styles an element when it is being activated (e.g., clicked).

```
button:active {
  background-color: #ff6347;
}
```

3. **:focus**: Selects and styles an element when it has focus.

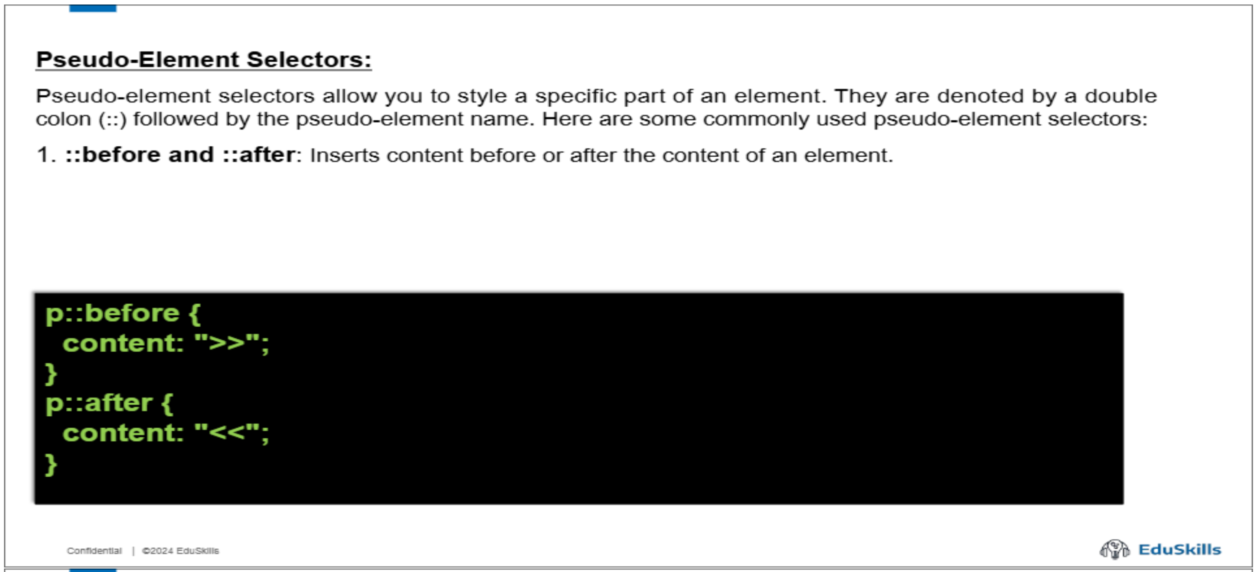
```
input:focus {
  border: 2px solid #007bff;
}
```

4. **:nth-child()**: Selects and styles elements based on their position in a parent container

```
li:nth-child(even) {
  background-color: #f2f2f2;
}
```

5. **:not():** Selects elements that do not match a specified selector.

```
input:not([type="submit"]) {
  border: 1px solid #ccc;
}
```

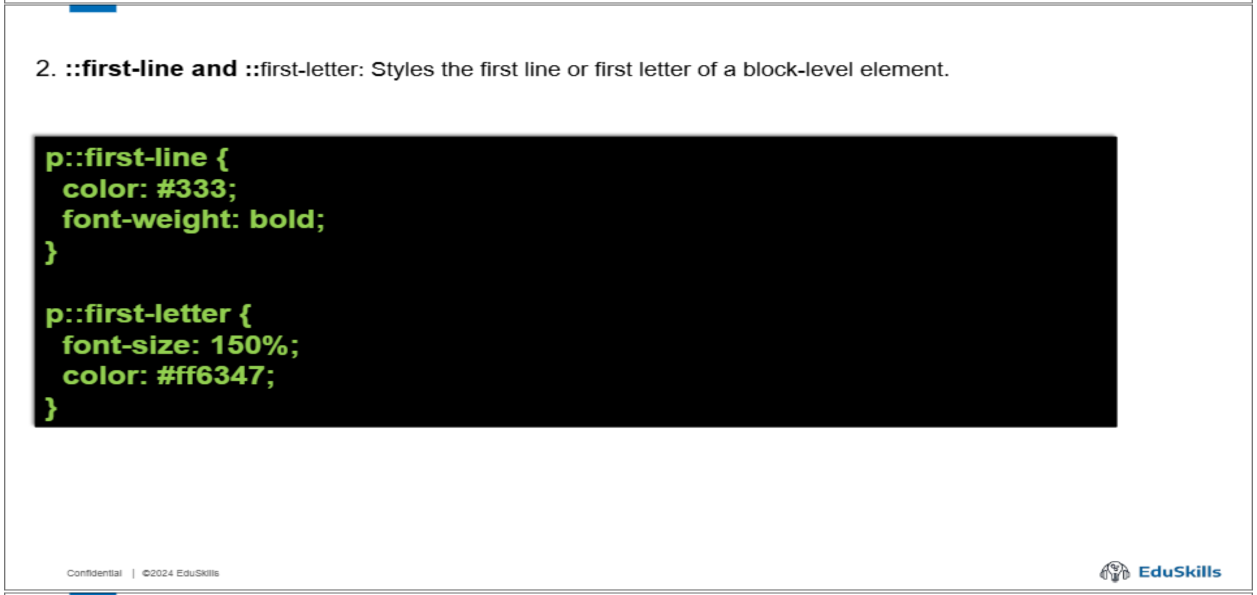


Pseudo-Element Selectors:

Pseudo-element selectors allow you to style a specific part of an element. They are denoted by a double colon (::) followed by the pseudo-element name. Here are some commonly used pseudo-element selectors:

1. **::before and ::after**: Inserts content before or after the content of an element.

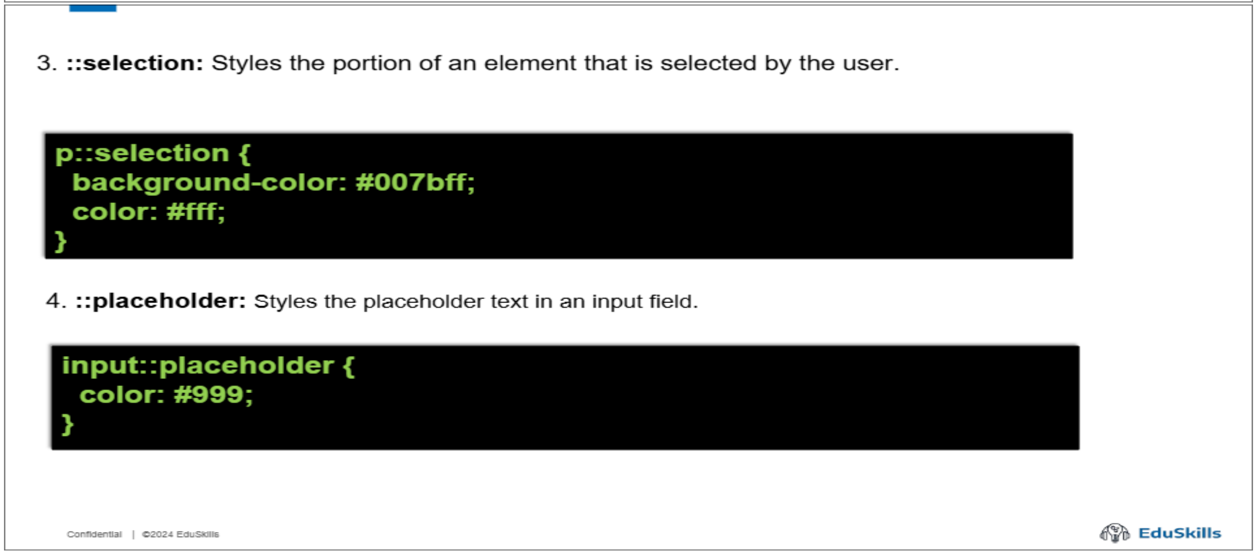
```
p::before {
  content: ">>";
}
p::after {
  content: "<<";
}
```



2. **::first-line and ::**first-letter: Styles the first line or first letter of a block-level element.

```
p::first-line {
  color: #333;
  font-weight: bold;
}

p::first-letter {
  font-size: 150%;
  color: #ff6347;
}
```



3. **::selection:** Styles the portion of an element that is selected by the user.

```
p::selection {
  background-color: #007bff;
  color: #fff;
}
```

4. **::placeholder:** Styles the placeholder text in an input field.

```
input::placeholder {
  color: #999;
}
```

Combining Pseudo-classes and Pseudo-elements:

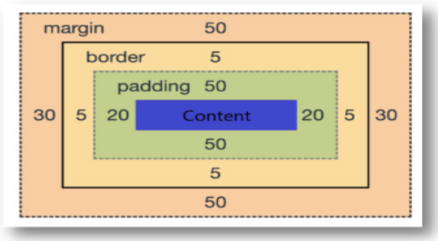
You can also combine pseudo-classes and pseudo-elements to create more specific and targeted styles.

```
/* Example: Style the first paragraph inside a div only when hovered */
div:hover p::first-line {
  color: #ff6347;
  font-weight: bold;
}
```

CSS Box Model

Note: All HTML elements can be considered as boxes.

The concept of the "box model" in CSS pertains to the design and layout of elements. It involves a rectangular structure that envelops each HTML element. The fundamental components of the CSS box model include content, padding, borders, and margins. The diagram below visually represents the hierarchical arrangement of these elements within the box model.



Explanation:

- Content: This refers to the internal area of the box where text and images are displayed.
- Padding: It defines the space between the content and the border. Essentially, it clears an area around the content. The padding is transparent, allowing the background to show through.
- Border: A border surrounds both the padding and the content, providing a visual distinction. It defines the outermost boundary of the box.
- Margin: The margin is the transparent space outside the border. It clears an area around the border, separating the element from its neighboring elements or the outer container.
- The box model allows us to add a border around elements, and to define space between elements.

Adjusting the Dimensions of an Element

To accurately establish the dimensions (Width and Height) of an element across various browsers, it is crucial to comprehend the workings of the box model.

Illustration

Consider this <div> element with a combined width of 350px and a combined height of 80px:

```
div {
  width: 320px;
  height: 50px;
  padding: 10px;
  border: 5px solid
gray;
  margin: 0;
}
```

Explanation:
320px (width of the content area)
20px (sum of left and right padding)
10px (sum of left and right borders)
= 350px (total width)
50px (height of the content area)
20px (sum of top and bottom padding)
10px (sum of top and bottom borders)
= 80px (total height)

For accurate measurement:

The total width of an element is derived by adding the width, left padding, right padding, left border, and right border:

Total element width = width + left padding + right padding + left border + right border

Similarly, the total height of an element is computed by adding the height, top padding, bottom padding, top border, and bottom border:

Total element height = height + top padding + bottom padding + top border + bottom border

Explanation:
320px (width of the content area)
20px (sum of left and right padding)
10px (sum of left and right borders)
= 350px (total width)
50px (height of the content area)
20px (sum of top and bottom padding)
10px (sum of top and bottom borders)
= 80px (total height)

More about CSS Height and Width:

1. Height Property:

The height property is used to set the height of an element.

It can be specified in various units such as pixels, percentage, em, rem, etc.

2. Width Property:

- The width property is used to set the width of an element.
- Similar to height, it can be specified in pixels, percentage, em, rem, etc..

```
div {
  height: 100px;
  width: 200px;
}
```

3. Min-Height and Min-Width:

min-height and min-width properties can be used to set the minimum height and width of an element, ensuring it doesn't shrink beyond a certain point.

4. Max-Height and Max-Width:

max-height and max-width properties can be used to set the maximum height and width of an element, preventing it from growing beyond a certain point.

Sample Code:

```
/* Applying height, width, min-height, min-width, max-height, max-
width to a container */
.container {
  height: 200px;
  width: 300px;
  min-height: 100px;
  min-width: 200px;
  max-height: 400px;
  max-width: 500px;
}
```

Important Points:

- The specified height and width values can be overridden by content size and other factors.
- Using percentages for height and width is often responsive, adjusting to the parent container.
- Be mindful of the box model (content, padding, border, margin) as it affects the final dimensions.

Table Styling with CSS:

- border-collapse: Specifies whether table borders should be collapsed into a single border or not.
- border: Sets the border properties for table cells.
- padding: Adds space inside the cell to separate content from the cell border.

```
table {
  border-collapse: collapse;
  width: 100%;
}
th, td {
  border: 1px solid #ddd;
  padding: 8px;
  text-align: left;
}
```

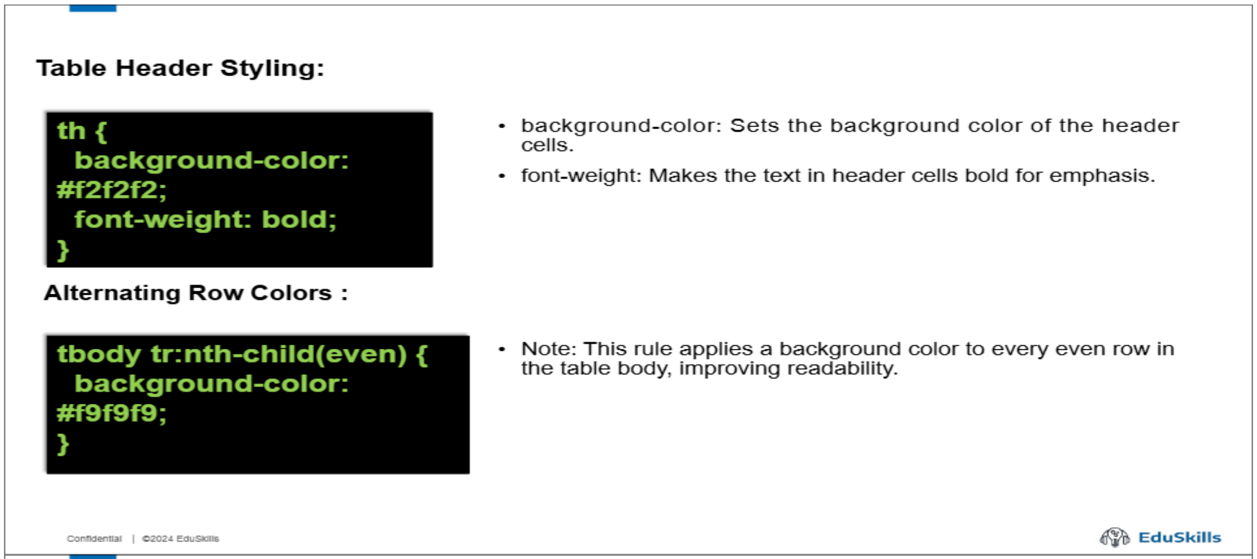


Table Header Styling:

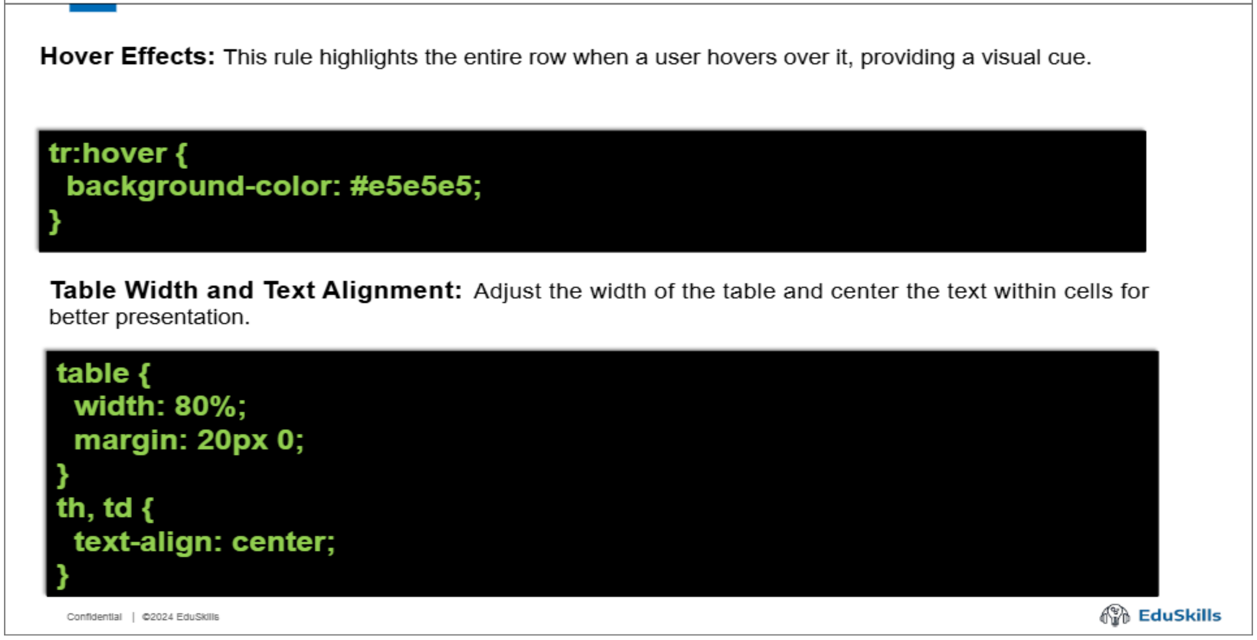
```
th {
  background-color:
#f2f2f2;
  font-weight: bold;
}
```

- background-color: Sets the background color of the header cells.
- font-weight: Makes the text in header cells bold for emphasis.

Alternating Row Colors :

```
tbody tr:nth-child(even) {
  background-color:
#f9f9f9;
}
```

- Note: This rule applies a background color to every even row in the table body, improving readability.



Hover Effects: This rule highlights the entire row when a user hovers over it, providing a visual cue.

```
tr:hover {
  background-color: #e5e5e5;
}
```

Table Width and Text Alignment: Adjust the width of the table and center the text within cells for better presentation.

```
table {
  width: 80%;
  margin: 20px 0;
}
th, td {
  text-align: center;
}
```

Responsive Tables: This media query ensures that the table becomes full-width and cells stack vertically on screens with a width of 600 pixels or less.

```
@media (max-width: 600px) {
  table {
    width: 100%;
  }
  th, td {
    display: block;
    width: 100%;
    box-sizing: border-box;
  }
}
```

Point to remember:
By combining HTML and CSS, you can create visually appealing and responsive tables that enhance the user experience on your website. Experiment with these styles and adapt them to suit the specific design requirements of your project.

Working with Color:

Colors are specified using predefined color names, or RGB, HEX, HSL, RGBA, HSLA values.

Color Property: The color property sets the color of text. It can take color names, hex codes, RGB, RGBA, HSL, or HSLA values.

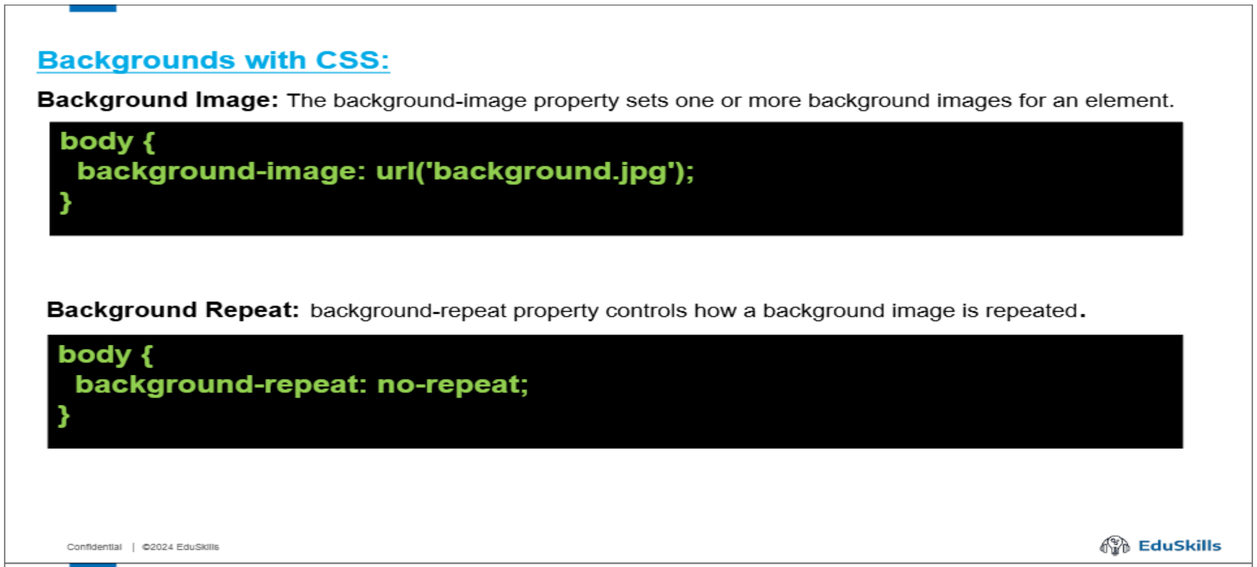
```
body {
  color: #333; /* Hex code */
}
```

Background Color: The background-color property sets the background color of an element.

```
body {
  background-color: #f4f4f4;
}
```

Opacity: Use rgba or hsla to set the color with an alpha (transparency) value.

```
.transparent-box {
  background-color: rgba(255, 0, 0, 0.3);
}
```



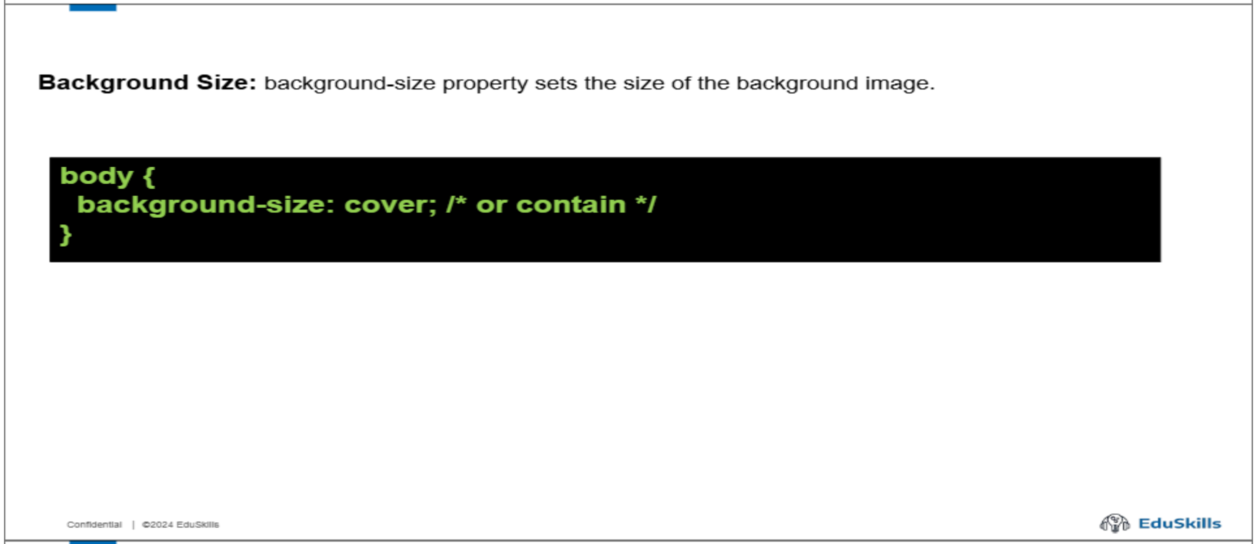
Backgrounds with CSS:

Background Image: The background-image property sets one or more background images for an element.

```
body {
  background-image: url('background.jpg');
}
```

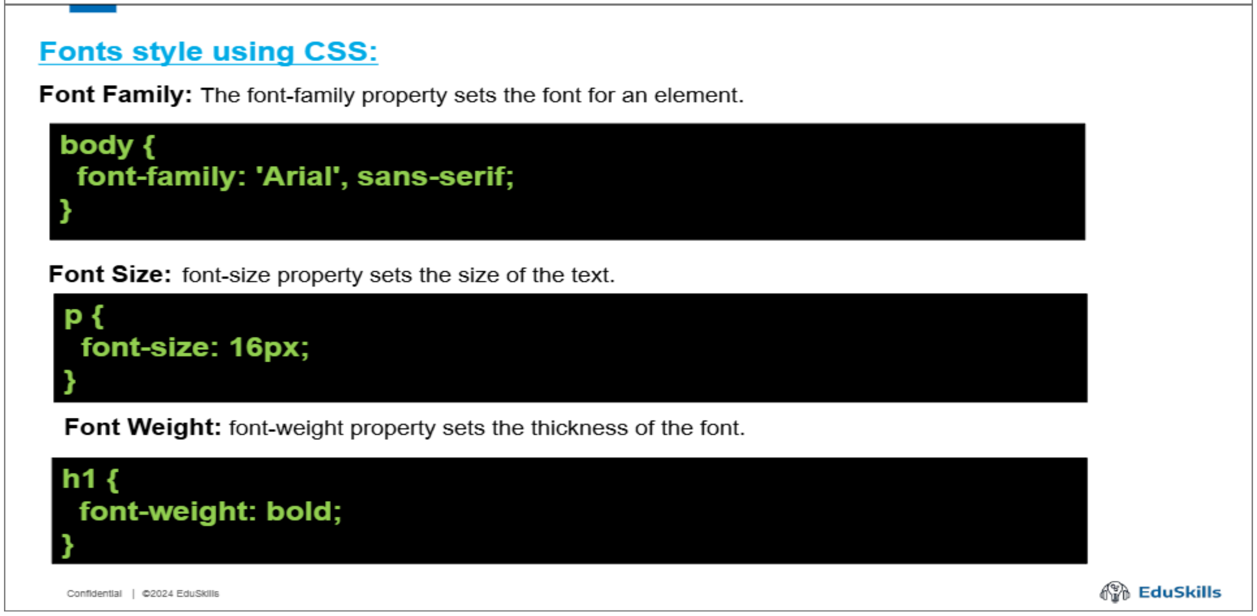
Background Repeat: background-repeat property controls how a background image is repeated.

```
body {
  background-repeat: no-repeat;
}
```



Background Size: background-size property sets the size of the background image.

```
body {
  background-size: cover; /* or contain */
}
```



Fonts style using CSS:

Font Family: The font-family property sets the font for an element.

```
body {
  font-family: 'Arial', sans-serif;
}
```

Font Size: font-size property sets the size of the text.

```
p {
  font-size: 16px;
}
```

Font Weight: font-weight property sets the thickness of the font.

```
h1 {
  font-weight: bold;
}
```

Text styling using CSS:

Text Alignment: The text-align property sets the horizontal alignment of text.

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

Text Decoration: text-decoration property decorates the text (underline, overline, line-through).

```
a {
  text-decoration: none;
}
```

Text Transform: text-transform property controls the capitalization of text

```
h2 {
  text-transform: uppercase;
}
```

- ## Image CSS:

Image Size: The width and height properties set the size of an image.

```
img {
  width: 100px;
  height: 100px;
}
```

Image Border: border property adds a border around an image.

```
img {
  border: 2px solid #ddd;
}
```

Image Opacity: opacity property sets the transparency of an image.

```
img {
  opacity: 0.7;
}
```

Using float in CSS:

The float property is used to define the alignment of an element within its containing element. It was originally designed for text wrapping around images, but it's commonly used for creating multi-column layouts and other complex designs.

Syntax:

```
selector {
   float: left | right | none | inherit;
}
```

Values:

- left: The element is floated to the left.
- right: The element is floated to the right.
- none: The element is not floated (default).
- inherit: The element inherits the float value
- from its parent.

EXAMPLE CODE:

```
/* Float an element to the left */
.float-left {
   float: left;
}
/* Float an element to the right */
.float-right {
   float: right;
}
```

Using display in CSS:

The display property defines how an element is displayed. It can alter the default behavior of an element, making it block, inline, or other display types.

Syntax:

```
selector {
   display: value;
}
```

Common Values:

- block: The element will generate a block-level box.
- inline: The element will generate an inline box.
- inline-block: The element will generate an inline-level block container.
- none: The element is not displayed.
- flex: The element becomes a flex container.
- grid: The element becomes a grid container.

EXAMPLE CODE:

```
/* Create a block-level
element */
.block {
   display: block;
}
/* Create an inline-level block
container */
.inline-block {
   display: inline-block;
}
```

Using position in CSS:

The position property is used to control the positioning of an element within its containing element.

Syntax:

```
selector {
   position: static | relative | absolute | fixed |
sticky;
}
```

Common Values:

- static: Default value. Elements are positioned according to the normal flow of the document.
- relative: Positioned relative to its normal position.
- absolute: Positioned relative to its nearest positioned ancestor (if any), otherwise relative to the initial containing block.
- fixed: Positioned relative to the browser window or the device screen.
- sticky: A hybrid of relative and fixed positioning. The element is treated as relative positioned until it crosses a specified point, then it is treated as fixed positioned.

EXAMPLE CODE:

```
/* Position an element
absolutely */
.absolute {
   position: absolute;
   top: 20px;
   left: 30px;
}
/* Make an element sticky */
.sticky {
   position: sticky;
   top: 0;
```

Debugging CSS:

CSS debugging involves identifying and resolving issues that arise when unexpected layout results occur, helping to pinpoint and address potential problems in your styling.

CSS Debugging Techniques:

Browser Developer Tools:

Using browser developer tools to inspect elements.

Modifying styles in real-time for testing.

Identifying and understanding the CSS box model.

CSS Linting:

Using tools like CSSLint to automatically detect and report common CSS issues.

Integrating CSS linting into development workflows.

Validation:

Validating CSS code using online validators.

Fixing validation errors and warnings.

Console Logging:

Using console.log() in JavaScript to debug dynamic styles.

Inspecting computed styles.

Cross-browser Testing:

Identifying and fixing cross-browser compatibility issues.

Tools like BrowserStack for testing in various browsers.

Creating flexible and fluid layouts

Flexible and fluid layouts is a crucial aspect of responsive web design, ensuring that web pages adapt to different screen sizes and devices.

1. Introduction to Fluid Layouts:

Definition: Fluid layouts use relative units like percentages for widths instead of fixed units like pixels, allowing content to adapt to different screen sizes.

Benefits: Better responsiveness, improved user experience across devices.

2. CSS Units for Fluid Layouts:

Percentage (%): Specifies a percentage of the containing element.

Viewport Width (vw): Represents 1% of the viewport's width.

Viewport Height (vh): Represents 1% of the viewport's height.

Flexible Box Layout (flex): Enables a flexible box model for layout design.

3. Media Queries:

Definition: Media queries allow you to apply styles based on the characteristics of the device, such as screen width, height, or orientation.

Syntax: @media screen and (max-width: 600px) { /* styles for screens up to 600px width */ }

4. Responsive Images:

max-width Property: Ensures that images do not exceed the width of their containing element.

Image Replacement Techniques: Using max-width, width: 100%, and height: auto to maintain aspect ratios.

5. Flexible Grid Systems:

CSS Grid Layout: Provides a two-dimensional grid system for layout design.

Bootstrap Grid System: A popular framework with a responsive grid system.

6. Fluid Typography:

Viewport-Relative Typography: Using vw or vh units for font sizes.

Calc() Function: Allows combining different units and values for more complex calculations.

7. Flexbox for Flexible Layouts:

Introduction to Flexbox: A powerful layout model for designing complex web layouts.

Flex Container and Flex Items: Understanding the container and items in a flexbox layout.

Flex Properties: flex-grow, flex-shrink, flex-basis, etc.

8. Examples and Case Studies:

Responsive Navigation Menus: Creating menus that adapt to different screen sizes.

Multi-Column Layouts: Designing multi-column content that adjusts gracefully.

9. Best Practices:

Mobile-First Approach: Start designing for small screens and progressively enhance for larger screens.

Testing and Debugging: Using browser developer tools to inspect and debug responsive designs.

Note: Creating flexible and fluid layouts involves a combination of relative units, media queries, and layout techniques. Mastering these concepts will empower you to build responsive and user-friendly web designs.

Working with CSS using web developer tools

Working with CSS using web developer tools involves using browser developer tools to inspect, debug, and modify styles in real-time.

1. Web Developer Tools:

Web developer tools are built into modern browsers and allow developers to inspect, debug, and modify web pages.

Common browsers with developer tools: Chrome Developer Tools, Firefox Developer Tools, Safari Developer Tools.

2. Opening Developer Tools:

Right-click on an element on a webpage and select "Inspect" or press Ctrl + Shift + I (Windows/Linux) or Cmd + Opt + I (Mac) to open the developer tools.

3. Elements Panel:

Displays the HTML structure of the page.

Hover over elements to highlight corresponding parts on the page.

4. Styles Panel:

Shows the CSS styles applied to an element.

Allows you to toggle styles, add new ones, and experiment in real-time.

5. Box Model:

Inspect an element's padding, margin, border, and dimensions in the "Styles" panel.

Understand the box model to control layout effectively.

6. Live Editing:

Change CSS properties in the "Styles" panel to see real-time updates on the webpage.

Experiment with different values and effects.

7. Adding New Styles:

Use the "+" button in the "Styles" panel to add new CSS rules.

Apply styles directly to the selected element.

8. Console Panel:

View error messages, log output, and execute JavaScript commands.

Useful for debugging CSS-related issues.

9. Network Panel:

Inspect network requests, including CSS files.

Identify slow-loading stylesheets and troubleshoot network-related problems.

10. Sources Panel:

Debugging styles by setting breakpoints and stepping through CSS code.

Useful for understanding how styles are applied.

11. Search Functionality:

Use the search bar in the "Elements" or "Styles" panel to quickly find specific elements or styles.

12. Device Mode:

Simulate different devices and screen sizes to test responsive designs.

13. Color Picker:

Use the color picker tool to inspect and modify colors easily.

Here is the Sass study material rephrased:

SASS BASICS

What is Sass?

Sass stands for Syntactically Awesome Stylesheets. It is an extension of CSS that adds features like variables, nested rules, mixins, etc. Sass was created in 2006 by Hampton Catlin and developed by Natalie Weizenbaum. It is compatible with all versions of CSS. Sass helps reduce repetition in CSS and saves time. It is free to download and use.

Why Use Sass?

As stylesheets grow larger and more complex, Sass helps manage them better. It provides features not available in plain CSS like variables, nesting, mixins, etc. This makes writing CSS easier to write and maintain.

How Does Sass Work?

Browsers only understand plain CSS. So Sass code needs to be compiled into CSS using a Sass preprocessor before being used. This process of converting Sass to CSS is called transpiling. We give Sass code to a transpiler and get CSS code in return.

Sass File Type

Sass files use the .scss file extension.

Installing Sass

There are many ways to install Sass on Mac, Windows, Linux. Some apps are free while some are paid. Details are on the sass-lang.com website.

System Requirements for Sass

- Works on all operating systems
- Supported by all major browsers like IE 8+, Firefox, Chrome, Safari, Opera
- Based on the Ruby programming language

Sass Variables

Variables allow storing information to reuse later. Sass uses the $ symbol to declare variables: “$variablename: value; “

Variable values can be strings, numbers, colors, booleans, lists, null, etc.

Sass Variable Scope

Sass variables are only available in the nesting level where they are defined.

Using Sass !global

The !global flag makes a variable global, so it is accessible at all nesting levels.

Sass Nested Rules

Sass allows nesting CSS selectors like HTML.

Sass Nested Properties

Many CSS properties share prefixes, like font-family, font-size, font-weight or text-align, text-transform, text-overflow. Sass helps reduce repetition.

@import and Partials

Sass helps keep code DRY (Don't Repeat Yourself) by splitting code into smaller files that can be reused. These partial files can contain CSS snippets like variables, mixins, resets, colors, fonts etc.

Sass Importing Files

The @import directive in Sass works like CSS for including other files.

Here is the Sass study material rephrased:

The @import directive allows including the content of one file in another. Unlike CSS @import, Sass @import includes the file during compilation so no extra HTTP request is needed.

@import filename;

No file extension is needed, Sass assumes a .sass or .scss file. CSS files can also be imported. Any variables or mixins in the imported file can be used in the main file.

Sass Partials

By default, Sass transpiles all .scss files. To import a file without transpiling it directly, start the filename with an underscore. These are called partials.

@mixin and @include

Sass Mixins

Mixins allow reusing blocks of CSS code. They are defined with @mixin.

```
@mixin mixin-name {
  props
}
```

Mixins are included using @include:

```
selector {
  @include mixin-name;
}
```

Passing Variables to Mixins

Mixins accept arguments to pass in variables:

```
@mixin bordered($color, $width) {
  border: $width solid $color;
}
.myClass {
  @include bordered(blue, 1px);
}
```

Default mixin values can also be defined:

```
@mixin bordered($color: blue, $width: 1px) {
  border: $width solid $color;
}
```

Mixins are useful for vendor prefixes:

```
@mixin transform($property) {
  -webkit-transform: $property;
  -ms-transform: $property;
  transform: $property;
}
.myBox {
  @include transform(rotate(20deg));
}
```

@extend Directive

@extend shares a set of CSS properties from one selector to another:

```
.button-basic {
  // styles
}
.button-report {
  @extend .button-basic;
  // extra styles
}
```

- This avoids duplicating properties in HTML:
- <button class="button-report">
- Instead of:
- <button class="button-basic button-report">

Chapter 2: Design Optimisation

What is multidevice design?

Multidevice Design in CSS

With the proliferation of different devices like desktops, laptops, tablets, and phones accessing the web, designing websites that work well on multiple devices has become very important. Here are some key concepts for multidevice design using CSS:

Media Queries

Media queries allow you to apply different CSS styling based on characteristics of the device like width, height, orientation etc.

For example:

```
@media (max-width: 600px) {
  .class {
    color: red;
  }
}
```

- This will apply the red color only for screen widths less than 600px. Common breakpoints are used to create styles for mobile, tablet, laptop etc.
- **Responsive Units**
- Using relative units like percentages, em, rem, vw, vh instead of fixed pixels allows elements to resize across different devices.

Flexbox & Grid

Flexbox and CSS Grid allow creating flexible and responsive page layouts that adapt across devices. Flexbox is good for 1-dimensional layouts while Grid handles 2D.

Mobile First Design

This involves designing for mobile as the base experience first, then enhancing layout for bigger screens. It helps avoid overly complex desktop designs.

Accessible & Responsive Images

Use srcset and sizes for loading appropriate image files. Use semantic <picture> tag for art direction.

Content Structure with HTML: Use HTML5 semantic tags like <header>, <nav>, <main> etc to structure content. Helpful for screen readers and SEO.

@Rules, Media Types and Paged Media

CSS Rules

CSS rules consist of selectors and declarations. Selectors target HTML elements and declarations apply styling to those elements.

There are different types of CSS selectors:

Type selectors (e.g. h1, p) target elements by element name

Class selectors (e.g. .intro) target elements with a specific class attribute

ID selectors (e.g. #heading) target elements with a specific id attribute

Attribute selectors (e.g. [target]) target elements with a specific attribute or attribute value

Declarations are inside curly braces and consist of a property and value (e.g. color: red;)

Multiple CSS declarations can be grouped into a block called a rule set (e.g. p {color: red; font-size: 16px;})

CSS rules can be placed inline, internally within HTML, or externally in a separate .css file

The cascade determines which CSS rules override others based on specificity, order, and inheritance

Media Types

CSS supports different media types like screen, print, speech, etc. This allows styling content differently based on the output device.

The @media rule is used to apply styles conditionally based on media query criteria like screen width, device orientation, etc.

Common media types:

screen - for color computer screens

print - for printed documents

speech - for screenreaders/speech synthesizers

all - suitable for all devices

EXAMPLE:

```
@media screen and
(max-width: 600px) {

/* Styles for small
screens */

}
```

Paged Media

Paged media refers to content formatted for print pages, like books, printouts, etc.

CSS provides various features to control paged media styling:

page-break-before/page-break-after - force page breaks

page-break-inside - avoid breaks inside elements

widows/orphans - avoid single lines on pages

page size, margins - control page dimensions

page selectors, pseudo-classes (:left, :right) - style based on page position

Example:

```
@page {
  size: A4;
  margin: 1cm;
}

h1 {
  page-break-before: always;
}
```

Creating responsive web apps vs. native apps

With mobile device usage continuing to grow, developers face the decision between creating responsive web apps or native apps. Both have their advantages and disadvantages that should be considered when choosing which route to take.

Responsive Web Apps

Built using HTML, CSS and JavaScript. Can be accessed on any device through a web browser.

Offer a "write once, run everywhere" approach working across different platforms and devices. Don't need to be downloaded from an app store.

Can be slower and offer weaker performance than native apps, especially on lower-powered devices. Limited access to hardware features like camera, GPS, etc.

Multiple CSS declarations can be grouped into a block called a rule set (e.g. p {color: red; font-size: 16px;})

CSS rules can be placed inline, internally within HTML, or externally in a separate .css file

The cascade determines which CSS rules override others based on specificity, order, and inheritance

Progressive Web Apps (PWAs) help bridge the gap by offering native-like functionality using web technologies. Features like push notifications, offline access and home screen installation.

Easier and faster to develop. Easier to maintain and update since changes happen on the server-side.

SEO friendly since they are indexed by search engines. Can drive traffic to your site.

Native Web Apps

Developed specifically for a given mobile platform like iOS or Android. Need to be downloaded from app stores.

Offer highest performance, speed and ability to leverage device hardware like camera, GPS, etc.

Can access platform features unavailable to web apps like Touch ID, Apple Pay, etc.

Can be expensive to develop and maintain multiple native apps for different platforms.

App store guidelines and regulations to comply with. App approval delays can slow time-to-market.

Gain presence and visibility in app stores which are important distribution channels.

Choosing Between the Two

Native apps make sense for performance-intensive games and apps, or ones needing deep hardware integration.

Responsive web apps provide a faster way to get cross-platform apps to market. Good for content/service-focused apps.

Many apps take a hybrid approach with a web app that also has a native shell for distribution in app stores.

Factor in development costs, target users, features needed and availability of in-house skills.

Note: There are valid use cases for both responsive web and native apps. Assess your priorities like time-to-market, budget, user experience and capabilities needed to determine the best approach. Leverage the strength of the web for broad reach, combined with native apps where they provide the most benefit.

Deciding which screen sizes to support

With the proliferation of devices with different screen sizes used to access the web, an important consideration in web design is determining which screen sizes to optimize your site for. Here are some key factors to consider:

Mobile usage - Mobile (especially smartphones) now accounts for over half of web traffic globally. Ensure your site works well on smaller mobile screens.

Popular device screen widths - Consider usage statistics to see the most common widths like 360px, 375px, 414px for smartphones and 768px, 1024px for tablets. Designing for the most used widths will optimize experience.

Desktop sizes - 1280px and 1440px wide screens are popular desktop sizes. Ensure key pages work well horizontally without excessive scrolling.

Graceful degradation - Support core functions on smaller screens even if some secondary content drops below the fold. Don't cripple experience on mobile.

Scrolling - Accommodate reasonable vertical scrolling on mobile. Horizontal scrolling on any device is usually prohibitive.

Touch targets - Buttons and links should be large enough to be tapped easily on touch devices. At least 44px x 44px.

Prioritization - Ensure critical content and calls-to-action are above the fold. Users on mobile often do not scroll much.

Readability - Use legible font sizes appropriate for each screen width. More line length on desktop but keep line lengths short on mobile.

Testing - Physically test on real devices of various sizes. Test responsive design resizing. Don't rely only on emulators.

Analytics - Review usage data to see popular device sizes and adjust focus accordingly. This evolves over time.

Delivering content across devices

Understanding Responsive Web Design:

Definition: Responsive web design (RWD) is an approach to web design that makes web pages render well on a variety of devices and window or screen sizes.

Key Concepts:

Fluid Grids: Creating flexible grids that can adapt to different screen sizes.

Flexible Images: Ensuring images are scalable and don't break the layout.

Media Queries: Using CSS media queries to apply styles based on device characteristics.

Viewport Meta Tag:

Importance: The viewport meta tag is crucial for controlling the viewport's size and scaling on different devices.

Usage: <meta name="viewport" content="width=device-width, initial-scale=1.0">

CSS Media Queries:

Definition: Media queries enable the application of different styles for different devices or conditions.

Syntax Example:

```
@media only screen and (max-width: 600px) {
  /* Styles for devices with a maximum width of 600px */
}
```

Mobile-First Design:

Approach: Start designing for the smallest screens first and then progressively enhance for larger screens.

Advantages: Faster loading times on mobile devices, better performance, and improved user experience.

Flexible Layouts:

CSS Flexbox and Grid: Utilize flexible box and grid layout systems for creating responsive and dynamic page structures.

EXAMPLE:

```
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
}
```

Responsive Images:

Resolution Independence: Use max-width: 100% to ensure images don't exceed the width of their container.

Image Formats: Consider using responsive image formats like WebP.

Testing and Debugging:

Browser Developer Tools: Use browser developer tools to simulate various device sizes and test responsiveness.

Cross-Browser Compatibility: Ensure that your design works well across different browsers.

Frameworks and Libraries:

Bootstrap, Foundation, etc.: Explore popular responsive design frameworks that provide pre-built components and grids.

CSS Preprocessors: Consider using tools like Sass or Less for more maintainable and modular stylesheets.

Performance Optimization:

Load Time: Optimize images, minimize HTTP requests, and use efficient coding practices to improve load times.

Lazy Loading: Implement lazy loading for images and other non-essential resources.

User Experience Considerations:

Touch-friendly Design: Ensure that your design is touch-friendly for mobile users.

Content Hierarchy: Prioritize and present content based on its importance on smaller screens.

Progressive Web Apps (PWAs):

Benefits: PWAs provide a seamless user experience across devices and can be installed on users' devices.

Service Workers: Explore the use of service workers for offline capabilities.

Personalising content

Understanding Personalization:

Definition: Personalization in web design refers to the process of creating customized experiences for users based on their preferences, behavior, and demographic information.

Importance: Personalization enhances user engagement, increases conversion rates, and fosters a sense of connection between the user and the website.

Types of Personalization:

Content Personalization: Customizing the content displayed to users based on their interests, location, or past behavior.

User Interface Personalization: Adapting the layout, design, and elements of the user interface to suit individual preferences.

Product Recommendations: Suggesting products or content based on user behavior, purchase history, or preferences.

Email Personalization: Tailoring email communication to individual users based on their interactions with the website.

Methods of Personalization:

User Data Collection: Collecting and analyzing user data, such as browsing history, location, and preferences, through cookies, user accounts, or analytics tools.

Behavioural Tracking: Monitoring user behavior on the website to understand preferences and patterns.

Machine Learning Algorithms: Implementing machine learning models to predict user preferences and provide personalized recommendations.

Challenges and Considerations:

Privacy Concerns: Balancing personalization with user privacy concerns and ensuring compliance with data protection regulations.

Algorithmic Bias: Addressing biases in personalization algorithms to provide fair and unbiased recommendations.

User Consent: Obtaining explicit consent from users before collecting and utilizing their personal data for personalization.

Tools and Technologies:

Content Management Systems (CMS): Utilizing CMS platforms that offer personalization features or integrating third-party personalization tools.

Customer Relationship Management (CRM) Systems: Integrating CRM systems to gather and manage customer data for personalized experiences.

Personalization APIs: Leveraging APIs that specialize in content personalization to enhance website functionality.

Best Practices:

Start Small: Begin with basic personalization and gradually increase complexity based on user feedback and data analysis.

A/B Testing: Experiment with different personalization strategies through A/B testing to determine the most effective approach.

Transparent Communication: Clearly communicate to users how their data is being used for personalization and provide options for opting in or out.

Future Trends in Personalization:

Real-time Personalization: Implementing personalization in real-time to adapt to user behavior instantly.

AI-driven Personalization: Advancements in artificial intelligence leading to more sophisticated and accurate personalization algorithms.

END Of Module-2 CSS