fCC Notes
Tanishq Kumar

Responsive Web Design Certification

Basic HTML

The *<main>* html tag lets readers know where the main content of your page sits more easily. Doesn't actually affect content.

img tags don't have closing tags, but must include an *alt* attribute that is what you see when you hover over it for accessibility purposes.

the *a* in *a href* stands for anchor, as it's a anchor hyper reference element that anchors the user to another website via a hyper reference to that website.

the *target* attribute can be set to tell the browser where to open the link, and the = "_blank" symbolises that you're opening a new window to display the web page you externally linked to

to link things internally, use *<a href = "#insertIDhere"></a>* where the text after the hash is a unique id like hello-world, and then you add that unique ID to the element that you want this to link to as well so it knows where you want to link to, and use *href = "#"* to make a dead link

you can get forms you make using the *form* tag or input taken in via the *input* tag to actually submit to a server using HTML alone (though the server has to be set up, manipulated, in Python and SQL) using the attribute *action = "/URLtosendtohere"* where you'd then define a function in python in application.py that corresponded to that URL

you can make a submit button using *<input type = "submit"></input>* or by doing a button tag there with everything else remaining the same

to make an input field required, simply add the attribute *required* into the input tag, and it's one that doesn't need to be set = to anything, much like the *placeholder* and *disabled* attributes used in input tags too

when you take input via a *radio* type (checkbox) then you should (best practise for what tools you're going to use later down the line) wrap around them a <label>. make sure to give all radio buttons the same *name* attribute so that the browser knows they're part of a group and that the user should only select one of them. best practise is to add a *for* attribute to the <label> that matches the *id = ""* that you should give the radio button (in addition to the *name* attribute you keep the same across radio buttons that are in a group)

you can also have checkboxes as a type of input—similar to radio, except these can have multiple checked for a question as opposed to just one like radio buttons. same structure with *<label>* tags applies to this. you can add the *checked* attribute to either ra radio button or a checkbox so that it's checked by default (again, takes *nothing = ""*)

the <div> tag is a container for other tags, and is used for page organisation and structure. on its own, it does nothing, but has attributes we'll gradually learn about that can make it a powerful tool. it's the most commonly used html tag of them all!

the *head* tag is used for metadata, like *title*, *style* (css), *meta* and *link*

Basic CSS

when declaring style inline, make sure to end with a ; inside of the quotes

while you make style = "" the actual property you're declaring within that, like *colour* should be followed by a colon, the key in CSS, but when you're declaring at the top of a document in pure CSS in the <style> tags of the <head> then you can just ignore the quotes and stick with the colon and semicolon. when doing so, make sure to add {} around each element's (eg h2) style rules.

to generalise further, you can make a "class" in CSS at the top, and instead of demarking it as just "h2" followed by {} you'd have .className followed by {} and then use class = "classname" inside the tags in the future to imbue within those tags whatever property you defined in classname.

as well as color, you can change font-size and font-family (type)

you can also import Google Fonts for even more variety in the fonts that you use. to do so, just *link href* the URL for the font you're interested in and give it *rel = "stylesheet" type =*

"text/css". when writing the font name in the actual element, include the name of the font in space if there's a space within that font name like Times New Roman but not if there isn't. include a "degrade" font in commas after your first choice font that the browser will use if there isn't the first choice font downloaded on the computer/browser.

you can style image sizes by making a class that specifies width = 500px and then applying it in the img tag

you can specify a class with border-color, border-width, and border-style, and you can also give one element multiple classes by specifying <element class = "class1 class2">

you can round off borders by using "border radius" property within a styling class, in pixels or in %

as an alternative to using the "class" designation to give elements style, you can use "id" for specific elements (and call on these to insert dynamism via JS later)

an ID attribute has a higher precedence than the class attribute, and so if two conflicting ones are added to an element, the ID will be the one that's used, and you can use it within the <style> section using the # operator (as opposed to the "." of the class)

each html element is inherently like a little box to the browser, and you can control its aesthetic by altering its *padding*, *margin* and *border,* all CSS properties we can control

the padding is the distance between an element and its border, whereas the margin is the distance between the border of the element and surrounding elements. these can also be negative to make the element larger. even more, you can control the padding on each side of an element to off-centre it, and do the same with the margins.

and instead of specifying the padding-left, padding-right etc of each element in CSS, you can just write them clockwise in the CSS property definition like 10px 20px 30px 10px etc.

in addition to specifying blanket CSS for ID and Class attributes, you can also leverage the *[type = "x"]* parameterisation in the CSS document to style all elements of a particular type in a certain way

px, in and mm are all absolute units of size, whereas *em* and *rem* are relative units that are relative to the parent font's size, with subtle differences between them, that are often used to design responsive websites that adapt to screen size beautifully.

CSS involves *cascading* so that anything inside a tag (like body) that is imbued with certain CSS properties will *inherit* those properties and so any <h1> you have inside the <body> will have the same aesthetic properties inherently baked in

you can have conflicting styling properties a lot of the time in complex style sheets. and so a few important clash precedents to remember include the fact that class styles take precedence over cascading/inhered styles, and that class styles declared further down in the style sheet take precedence over ones declared further up when the two different classes are used on the same element. similarly, ID attributes take precedence over class attributes, and inline style declarations override all of classes, IDs, and inherited properties. to get one particular CSS style declaration to take precedence over all the others, though, you can just add *!important* to any style property and it will then always take precedence over any conflicting style properties.

you can also replace "color: red" with a hex code for that colour (just prefix with #) and you can shorten hex codes like #FF00FF to #F0F, and alternatively, can add colour using rgb(253, 202, 113)

you can create —cssVariables where you set them to a particular property (like color: and then set other properties equal to var(varName) and they will change dynamically whenever you change the property of that variable that you assigned them to. you can also assign a fallback value when you're setting other properties to the variable property in case the variable property is NULL or fails by doing *propertyX = var(—varName, black);*

some browsers can't interpret CSS variables, and so make sure to add fallback values so some users on old browsers don't have a shitty experience

in the same way as *body {}* in <style> makes things that apply throughout the body, using *:root{}* makes things throughout the entire HTML document. if you want to override a variable, then set the variable equal to a different value to the one you initially declared it and that will take precedence

you can make your site more aesthetically responsive by adding media queries like:

@media (max-width: 200px) { :root {} }

so that if the medium that the user is viewing the web site on is smaller than 200px, then the variable/property declarations you have in your actual :root{} will not be used, but the properties you define within the media query *will* be used

Applied Visual Design

IMPORTANT INFORMATION

This section is on UX design, encompassing typography, graphics, colour theory, layout and proportions, and more.

text-align is another important text presentation CSS property, with values being left, right, center, justify, and more.

you can use <strong> for bolding text, <u> for underlining it and <em> for putting it in italics, as well as the <s> tag to strike through text (all need closing tags)

use <hr>, a self closing tag that creates a horizontal line

instead of *color* to change the colour of the text or *background-color*, which changes the entire background I can use *background colour* inside the CSS *of a specific element* to give the text a background to make it more readable (using rgba() to adjust opacity)

when you adjust the height, weight or padding of an element, what you're really doing is adjusting the specifications of the box that that element is in naturally, even if it's invisible to begin with

you can create a shadow around an element using the box-shadow property, which has values of the x-offset, y-offset, blur radius (gradient starkness), and a colour value

opacity is another CSS property that you can cvary between 0 and 1, where 1 is absolutely opaque, and 0 completely see through

there is a text-transform property in CSS that can take values like *uppercase, lowercase, capitalise, or inherit (take same as parent element)*

much like font-size is for size of text, font-weight is for thickness (the *strong* tag is an abstraction for the browser implemented font-weight: bold; via an ID in the style sheet

line-height is a property that determines the height of each rectangle that is a line, can easily be used in paragraphs of text to vary spacing

a "selector" is just something that specifically styles/selects for a particular element in the HTML, for example, like class="" or id=""

a *pseudo-class* is a keyword you add to selectors in the style sheet to style for a particular state; for example, instead of a {} which would style all anchor tags naturally, you use the colon to do a: hover {} to specify the style that anchor elements take when they are *hovered over* (separate to their natural styling). in this way, you're selecting for more specificity (selecting a hovered anchor tag instead of just selecting an anchor tag) and so it's an extension of a class, and thus is termed a *pseudo-class*.

you can move individual HTML elements from their normal position by using the *position: relative* property (where you're shifting them relative to normal flow), coupled with a *top, bottom, left* or *right* properties (called offsets), to which you give the padding you want to shift the element by as a value, in pixels

<section> defines a thematic section in a web app, whereas <div> is just a bucket that carries no meaning on its own without identifying class, id, etc.

you can lock something in position using position: absolute, which locks it relative to its closest *positioned* parent (so if that moves, this will move too), and if there isn't one, then just locks relative to <body> that is, stays in place

you can also give the position property: the "fixed" value, which is the same as the absolute value, but the HTML element no long moves on the user's screen when they scroll, and the other elements are no longer aware of the "fixed" element's existence and so you may have to adjust other things accordingly. putting a position:fixed can be important in making animations work.

you can use *float* property to define where an element sits horizontally on the page, giving it values like *left* or *right*, coupling it with an appropriate width to be able to fit two elements side by side in two "columns", like by coupling a *section* and *aside* tag

you can use a *z-index* property when having a stack of overlapping elements created, with the higher z-index specifying which shows up on the "top" of the stack; that is, a z-index with value 1 would be specified in the class/ID/inline of an element in the very bottom of the virtual stack, “furthest” from the user

you can centre an block element (a shape, text, etc.) by setting its CSS *margin* property to “auto” which automatically adjusts the margin such that it fits perfectly in the middle, and you can do this with inline elements (images that normally just stay on the same line as whatever else is in where they’re declared) by manually changing their CSS *display* property to “block”.

complementary colours are defined as those, when combined, produce grey (and are thus “opposites” in terms of visual looks) and product a strong, beautiful contrast when coupled together

mixing two primary colours (RGB) gives a secondary color, like cyan, a mixture of blue and green, etc., and secondary colours are complementary to the primary colour *not* used in their creation (an interesting symmetry). tertiary colours come about by adding a primary colour to a secondary colour. you can use the “split-complement” colour design model by pairing a base colour with the two colours adjacent to its complementary colour on the colour wheel.

you can also use the hsl() model to come up with colours, where hue is the “color”, “saturation” is the amount of grey in a colour, and lightness is the amount of white or black in the colour, taking in three values—hue, which is the angle from 0-360 on the colour wheel, and the other two are percentages

adding white to a certain hue makes a “tint” of that colour whereas adding black makes a “shade” (X tints of blue, Y shades of blue)

a <nav> just groups lots of links that would be present in something like e navigation bar for accessibility and clarity reasons—no real functionality

you can use *background: linear-gradient(90deg, red, yellow, rgb(204, 204, 255));*

you can make interesting designs using the repeating-linear-gradient() property in CSS, where you give an angle, as well as several colours and the pixel lengths from the starting point at which you want those colours to peak eg:

```
repeating-linear-gradient(
        50deg,
        yellow 0px,
        blue 40px,
        green 40px,
        red 80 px,
)
```

would give you a blend angled at 50 degrees like a backslash, doing: 0px [yellow blends to blue which peaks at] 40px [green immediately starts and blends to red, which peaks at] 80px
and by manipulating these colour schemes instead of having gradients (eg if you changed blue to yellow you wouldn’t have a gradient) you can make interesting striping patterns.

you can also set the background: as a url(“www.pattern.com/coolpattern.html”)

you can change elements’ sizes by doing setting the transform: property to the function scale(scaleFactor) and combine this with other CSS tricks like feature:hover {}

the skewX(deg) and skewY(deg) functions can be values you set transform: to and the former simple tilts a shape by that amount and the latter rotates it.

the ::before and ::after are pseudo-elements that you use to modify an existing CSS element (like :hover) and the content: “” property it takes just inserts into the web-page that content before or after the element it’s associated with (like div::before { content: “hi” }) will print “hi” on the page before every div. an important application of this is in making shapes, like hearts, where we set the content to “”, and just manipulate other properties, and associate the pseudo-element with a shape (class/ID that we created), before moving them around with offsets of position. you could do the exact same thing by just creating two other classes but then you’d have to associate anything you wanted to be a heart with three classes instead of just one

in addition to making shapes, controlling colours and design and proportions as well as layout, you can animate with CSS! there are two parts to any animation: defining it via @keyframes AnimationName, and defining it via making a class that you call to execute it, wherein you define the AnimationName as a CSS property, and animation duration as a CSS property. in @keyframes animationName, you use 0% {} 50% {} and 100% {} etc. to control what the animation actually does—as if it were a movie and you were defining aesthetic properties for the beginning, middle, and end. An animation to make a rectangle to from blue to green to yellow might be:

```
  #rect {
    animation-name: rainbow;
    animation-duration: 4s;
  }

  @keyframes rainbow{
    0% {
      background-color: blue;
    }

    50% {
      background-color: green;
    }

    100% {
      background-color: yellow;
    }
  }

</style>
<div id="rect"></div>
```

animation-fill-mode is a property that you add to the class/ID declaration that specifies what happens when the animation finishes, and "forwards" is the value that means keep the property we animated to continuing into the future even after the animation finishes

you can modify the *position* values using offsets in animations to program in motion, by altering the pixel values of the *top* property etc.

the animation properties we know so far for CSS declaration are name and duration; others include animation-iteration-count which can be set a value of how many times you want to loop through that animation, which can take the value of infinite.

you can change the keyframes % at which the same animation occurs for different things to get them to go through the same (or different!) animations at different (non-synchronous) rates. this can also be done by varying animation durations for each animation.

the animation-timing-function is another animation property, but one that controls *how* the animation varies over the duration you defined (when it peaks etc.), taking values like *ease* (default) which is ease-in + ease-out, ease-in for slower beginning, ease-out for slower end, and linear for constant throughout

you can leverage *even finer* control over how animations play out, temporally speaking, using *Bezier Curves* which are almost like literal mathematical graphs on a 1x1 grid, with 4 natural points. the x-axis is time and the y-axis is animation progress. the first and last points are obviously fixed as 0,0 and 1,1, but the middle p1 and p2 can be varied by manipulating their x,y co-ordinates, thus changing the rate of animation with a lot more control than merely with "ease-in" etc. the cubic-bezier(x1, y1, x2, y2) input is a value you plug into the animation-timing-function:

you can also set the y values >1 counter-intuitively to mimic it being at "1.6" times through the cycle by some time value from 0 to 1 (super-sped up, if you will)

<u>Applied Accessibility</u>

when text is being used purely for decoration, and in other similar scenarios, you don't need to include an alt="" tag since deaf people don't need to hear "blue box for aesthetic purposes" in their assistive technology

<h1, 2…> tags are used for web page indexing and for people with accessibility difficulties—you should use higher tags like <h3> to start and delicate sub-sections of <h2> tags and so on. you should only ever use <h1> tags to demarcate the meaning/main message of the whole page, as search engines will look at that. to change the relative sizing of these, just use CSS.

you can use <article> to group some chunk of text that can stand on its own, while <section> groups *thematically related content* and <div> just adds structure to the page through the selector tags it employs

people with assistive technologies use the inbuilt structure we create using these tags to navigate around the page

<header> is what goes inside <body>—quite separate from <head>, which comes before body (obviously) and contains page metadata—and it's where you add the generic content at the top of each page (the nav bar, for example)

clean structures not only make things easier for you front-end engineers, but also allow those with assistive technologies to more easily parse your content since many of these technologies look for the keyword tags to understand why the page is really about. your navigation bar would go inside <nav> tags within the <header> tag within the <head> tag

the <footer> landmark element—again, within the <body> tag—is used to contain all the copyright information you'd normally have on the page at the bottom—for clarity's sake; none of these really convey any meaningful information except for helping with code readability and maintainability (which is crucial if your project scales!)

you add an <audio id ="" controls> tag around your audio <source src="" type="audio/mpeg"> tags because then it makes it clear that the link the disabled person cannot access contains audio. the controls attribute in the <audio> tag simply add keyboard functionality for users listening to the audio

you add <figure> tags before and after your figure, and within those (but after the figure), you can add <figcaption> to caption the figure for disabled users.

<label> can have the "for" attribute, where you write what the text corresponding to some input corresponds to. for example, you'd do <label for: "name"> Name: </label> to make it clear that the option is selecting a name value when disabled readers access it. make sure this corresponds to and matches with the "id" part of the <input> tag

<input> tags can take the type attribute which can have the value "date", as well as the usual others (text, radio, checkbox, etc.)

for any html element, you can add the accesskey = "" attribute for keyboard-only users to quickly use that functionality (like a submit button or the like)

you add the attribute tabindex="" to determine whether an element can be focused or not by the keyboard, with 0 being the value that means it can be reached with a keyboard and tabbing to move through the fields of input work, and -1 meaning it cannot and they don't. adding tab index 2,3 and so forth means that after "1", pressing tab will take you straight to those fields (but not necessarily down the page, which would make for bad UX)

:focus is a cool pseudo class that executes certain CSS properties when the div of that element is clicked on (like background-color=yellow)

Responsive Web Design (for different devices—mobile optimisation is key)

IMPORTANT INFORMATION

remember media queries, where certain CSS is applied if you have a viewport of a certain size:

```
    @media (max-width: 200px) {
        css here
    }
```

to make an image responsive to screen change, just add a certain CSS selector that does these:
max-width: 100%; (this sets it to the original width as a proportion of container on screen size)
display: block (this makes it a block element that has its own line as opposed to inline)

height: auto; (this maintains the aspect ratio as the screen size changes since you've kept width fluctuating depending on screen size, this then does too)

you make images appear clearer by using them at half the size they originally were (use proportions where you can)

you can size text relatively by using viewport units—styling a font size as width: 2vw (viewport width units in 2 percent) and height the same way with vh, and vmin as a fraction of the viewport's smaller dimension

//IMPORTANT—WHEN YOU MASTER THESE YOU DON'T NEED BOOSTRAP

<u>CSS Flexbox</u>

CSS Flexbox is a way of using CSS to make all divs and containers flexible to change easily in standard ways for different screen sizes and dimensionalities, creating a dynamic UI. it allows you to manipulate content in one direction.

for you to use flex box properties via CSS to style selectors, you must enable them by adding a property display with a value of flex—this automatically expands things to take up free space around them or to move away from overflow by themselves.

you can specify CSS properties of a certain name that will only take on those values when in a certain section, like:

```
header.value {
        CSS properties here
}
```

and then when you have something like <header><p class="value">ting</p></header> it'll take on the values you set up before, but doing class="value" *outside of header will not do anything*.

if many things have inherited the "display: flex;", you can manipulate them using the flex-direction attribute, which controls tabling. for example, by setting that equal to row, column, or row-reverse or column-reverse, you can manipulate the layout of everything with that property's relationship with each other in a table-sort of look.

whenever you create a container (via div) and imbue it with the property of being a flex box (display: flex;) and things inherit that property and are component members of the div, you can manipulate how they are arranged in space using justify-content: attribute, which can take values of centre (all children are in center, next to each other), flex-start (at the beginning of the flexbox), flex-end, space-between (split between either end of it) and space-around (in the middle with space on either side of them). you'd use flex box when you want to lay out a collection of items in a certain way and want a certain spacing between them (like in a nav bar!). justify-content attribute aligns things along the *main axis (in the parent whether you've specific flex-direction: column* or whatever)

you can align things to the main-flexbox axis using *justify content* attribute, but you can also align things to the cross axis (the opposite of the main axis—so a column, by default) and set the *align-items* attribute to values to move the entire row/column you set in the main axis up or down or left or right using values like flex-start, flex-end, stretch, baseline (of text).

flex-shrink and flex-grow are attributes you give to the children components of a container so they grow with the container as it's resized—they take small integer inputs

you can wrap items within a container using flex-wrap, which is an attribute that can take the values wrap, nowrap and wrap-reverse (all fairly intuitively named)

flex-basis is the attribute that controls the initial size of an element, and is written in em units, which are relative units that are fractions of the font-size of the element (or its parent, etc.). flex basis seems similar to width? it's different in that it 1) only applies to flexed items 2) takes the direction of the flex-direction of the parent (so it's not necessarily always width), though it should visually look the same as width when rendered on a browser

you can manipulate flex grow, shrink and basis all in one line of CSS shorthand that is native to the language. you use flex: 1 0 auto; as the default (makes sense) and can change those numbers to customise much easier than manually changing width for different UIs.

you can specify which order flex components come in the flex container using the "order" attribute which can take on integer values, with larger integers suggesting they come later. the default is in the order you write the source html in.

the align-self property takes the same values as align-items, but is meant for individual children components instead of the parent component, and takes precedence over the parent component's alignment. it's so you have more granular control over how things are aligned.

<u>CSS Grid</u>

Grid involves another set of tags and tools that allow you to convert any container into a two dimensional grid, allowing you to more easily manipulate the layout of elements in two dimensions.

you define it in the same way as flexbox—dispay: grid;

you set up columns in your grid by doing grid-template-columns: width1, width2;

grid-template-rows work the same way

you can quantify the widths using auto, px, em, fr (how many fractions of the available space), %, and auto

grid-column-gap adds a gap between the columns you create, and grid-row-gap works functionally identically

grid-gap is shorthand for both of these, if you give it one value it'll assign it to both, and otherwise the first to rows and the second to columns.

so far, all these CSS properties have been used to manipulate the grid itself, and not really the *items within* the grid (container). to manipulate the grid items, a property to add to an item is *grid-column*, which takes values that correspond to the *grid lines you want that item to span, vertically.* so if you imagine a 3x3 grid broken into 4 vertical and 4 horizontal lines, saying grid-column: 4/3 means your item only spans the last column of the grid (but still one row). *grid-row* is an analogous property that allows you to control the row span of the item.

to modify how much of the width of each cell an item occupies, use justify-self property, which is set to "Stretch" value by default (the whole width), but can also take the values of start, end, and centre (which only give the actual item a width corresponding to auto—but move that width accordingly around the cell)

in the same way that justify (horizontal usually) works, align-self will move an item up or down within its cell.

you can add justify-items and align-items to make this horizontal or vertical alignment happen with all the items in a given contain. remember that this takes the same values as the above; that is, stretch, start, center, end.

you can group parts of your grid into custom areas for clarity and structure, using this syntax:

```
grid-template-areas:
        "header header header"
        " ad      content    ad"
        " footer content footer"
```

where the individual items represent cells and the quotation marks represent rows in the grid. in this context, you'd use a period to represent an empty cell.

the reason doing the above is useful is because then you can create a custom selector for the grid areas you laid out, like .someItem {grid-area: header OR grid-area: ad; }will style accordingly just for the specific rows or columns selected within those.

you can use the above functionality without actually explicitly creating a grid-areas-template in the parent container, too. by saying .someItem { grid-area: 1/1/3/4; } you demarcate the fact that your style will make the item go from row 1 to 3 and column 1 to 4

you can use repeat(numberOfRepeats, valuesToRepeat) instead of repetitively typing out 100 columns of alternating 50px 2fr in the grid-template-columns/rows property.

you can define a column's width value in grid-template-column: minmax(50px, 20%) to allow for it to resize as the interface changes size.

you can set the row or column amount to "auto-fill" whereby the number of rows/columns will automatically be adjusted depending on the size of the screen. so you'd use grid-template-columns: repeat(autofill, 50px); to get variable amounts of columns, of 50px width each.

autofit is a value that's functionally identical to auto*fill* except for the fact when the container its items are in gets so big, it doesn't add new columns, but resizes your current ones to fill up the container!

you can nest all of these grid and flex box functionalities into a media query to create responsive web sites from the ground up.

aside from 1 vs 2D, grid is *layout first* (if you want to make a header, you have to set up the columns and rows and manipulate those to set it up before adding conte) whereas flex box is *content first* (you simply add the display:flex; property to existing content and the styling is done around the content, and you manipulate the content, not arbitrary divisions you create).

Javascript Data Structures & Algorithms Certification

js basics

null, undefined are actually data types in javascript

to escape a string inside a string problem, you need to \"type stuff like this\"

you can escape more than just a string using \, you also use that for newline, tab, return, etc. within a string if you want it formatted a certain way

everything is an object in jS, and you have lots of innate methods built in, for example for objects that have string properties you can innately access .length() property

you can treat strings in jS like arrays of characters, and access particular characters via []

strings in jS are *immutable*, and can't be modified, only changed entirely

you can use push(), pop(), shift() which pops the first element in an array, unshift() which adds elements to the beginning of the array as opposed to push() which is at the end

create a function literally by defining *function functionName (param1, param2) { }*

if you just define a function, without using *var* then it's a global function by default whereas var is definitionally local in its scope.

you can have local and global variables with the same name—the local ones take precedent within a function (and naturally aren't defined as local outside the function due to local function scoping)

the switch function is built-in and can be used instead of many if-statements, in the following manner:

```
        switch(val) {
                case 1:
                case 2: //1 and 2 will do the same thing
                        statement;
                        break;
                default:
                        statement;
                        break;
}
```

typecasting means taking something that is a certain data type and temporarily casting it as another data type.

you make a jS object via:

```
        object = {
        "heads": 1;
        "legs": "three";
}
```

you can access these by then doing object.heads or object["heads"] and you can create new properties the same way you'd access them, and you can delete properties simply using *delete object.property;*

test whether a certain object has a property via ".hasOwnProperty"

the reason JSON gets its name is because it literally resembles how you'd store data in a jS object you can have more complex variables and data structures by nesting these—say, an array of objects, with arrays nested inside of them, and more.

—> make sure to understand the spec of a problem very well from the get go or you'll spend ages just trying to figure the details when you don't fully understand the bigger picture

for ([initialization]; [condition]; [final-expression]) {}

whenever you're dealing with js objects, and you want to access a property by a variable name, use bracket notation—its use is not identical to dot notation, it is more general, even if less convenient.

use Math.random() to generate a random decimal between 0 and 1 (then can shift using +/- and Math.floor() as well as Math.ceil())

parseInt(string) = int, and takes the optional argument radix, which is the base you want the answer returned in

the faster version of the if-else statement is called the "Ternary operator" and can be written and take this form:

condition ? statement-if-true : statement-if-false;

you can even chain these for even more efficiency by adding another colon and continuing the same way within the statement-if-false condition

es6

let essentially declares a value that cannot be changed once created—but you can have different values under the same variable name depending on the scoping if you use let within a block and then a function, you can access two different values under the same name. because many people making large applications were essentially accidentally changing variables they didn't mean to without being thrown an error—making it very hard to debug where their program is going wrong (which is why most large companies don't use weakly typed languages).

another difference between *var* and *let* is scope. a *var* will have global scope if declaring outside a function, or local scope if declared inside the function call. even when created for a *for* loop, i will stay globally.

by convention, consts are written UPPER_CASE and mutable variables in camelCase, with arrays in all lowercase.

importantly, you can still mutate consts, just not redefine them completely. this is why people will use const and let for most of their variables, to make value changes more deliberate and strict so debugging becomes that much easier.

use Object.freeze(name) to prevent an object from being changed *at all* (without an error if someone tries to change them)

syntactic sugar for anonymous functions involves arrow notation. you replace = function() {} with = () => {} and you can even simplify down to one line if you just have the anonymous function return something like this: = () => “value you want returned” and even omit the *return* !

even more powerful use:

// multiplies the first input value by the second and returns it
*const multiplier = (item, multi) => item * multi;*

you can set default values for you parameters within the argument of the function(name= ‘James’) definition

the rest parameter is …args which means a function can take infinite arguments, or an unspecified number, which are stored in an array *args.* common functions to use on this array involve .reduce(total, currentValueInArray) which execute some anonymous function to distill all the values in the array down to a single value.

the *spread()* operator is written the same way: …arr, and is used to make an array into a set of comma separated values, to feed into some function that only takes in CSVs or alternatively just to copy the values in the array to another one

destructuring syntax is a way to assign variables to object properties a more succinct manner: instead of object definition, variable1 = object.prop1, variable2 = object.prop2, you can do:
        const { var1, var2 } = object (make sure the names of var1/var2 match object declaration properties)
but you still have to define the object beforehand to actually insert meaningful values.

but if you want more versatility, where var1 and var2 don’t have to match the object definitions, you can do *const { prop1: var1, prop2: var2 } = object*

you can also destructure an array and assign variables to specific elements in the array conveniently using this syntax: *const [a, b,,,,c] = arrNums where a = 1, b = 2, c = 5*

you can even mix and use the spread operator with the destructuring syntax:
const [a, b, …arr] = numArr which will give a = 1, b = 2, arr = 3, 4, 5…

and you can even feed structuring assignment variables into function parameters:
const funcName = ({max, min}) => (max + min) // where max, min come from the stats object with properties stats.max and stats.min from above

you define methods within an object in the following way:
```
        const object = {
        name = ‘Bob’
        sayHello() {
                return “Hi, my name is ${this.name}!”;
        }
}
```

you can create *getter and setter* functions within an object declaration to hide internal implementation details. say you created:

```
class Book {
        constructor(author) {
        this._author = author;
        }
// getter
        get writer() {
        return this._author;
        }
// setter
        set writer(updatedAuthor) {
        this._author = updatedAuthor;
        }

}
```

so now when someone does Book.writer they get the author, but they don't know that the ._author bit exists and so the developer hides internal implementation details so the consumer can use an abstraction, which is helpful if you want to make something more difficult to reverse engineer.

you can export a function so it can be called in another program that contains *import { funcName } from ./fileName.js* using *export { funcName };* in your program. if you want to be able to call any function declared in the program, use *import * as nameYouWillCall from srcURL*. you can also add a *export default* property in front of one function in the file that you're importing for that to be the "fallback" value in case nothing in particular is imported/exported. then you call the default import using the name you want to use in the file you're importing to:

if you exported the function "additionFunction" by default, you might import *add from filename* and then go on to use add(), even though in the original file it's called additionFunction. Note that you didn't have to do import { add } since it knew the default exported function is what you were referring to.

in jS, *asynchronous* means not in order. so instead of executing a line, then executing the second line, which sends an AJAX request, and waiting for the response to come back, and then the third line, you send the request, execute the third line, and then process the AJAX data when it returns. This uses time more efficiently (doesn't just twiddle thumbs while waiting on a call) but does other important things during that time. you rarely have to write the async command implementation yourself, it's done under the hood when you use jQuery.ajax or things like that.

threading refers to parallel processing on the same CPU. while waiting on one action to return a value, the CPU switches and performs the next action in the code, often finishing that before the response from the first one came back in the first place. these two different "paths" were different threads of code. note how threading is an asynchronous process.

a constructor function is one that you use to give other functions certain properties determined by the constructor. For example, saying req = new XHttpRequest, where XHttpRequest is a constructor function.

Similarly, jS has many built-in constructor functions, such as a promise, where you can say:

```
my promise = Promise((resolve, request) => {
        if (request.recieved) {
        resolve(result);
};
        else {
        reject("bad");
};
});

mypromise.then(result =>{
	// do something with the result
});
```

this showcase how you'd use a promise to execute something asynchronously, where you send some sort of request to another API, wait on a response, and if you don't get it, throw an error message, and if you do, then execute some action with the data you received back from there server you made the request to—and all of this is naturally handled asynchronously to optimise for time efficiency.

the mypromise.then function is called a callback function, and similarly you can do *mypromise.catch(error => {})* to execute on the thing that you rejected in the promise outline

regex

a regular expression, or regex, is a templated series of characters you look for in another source—say a file, page, or source code:

```
let myString = "Hello, World!";
let myRegex = /Hello/;
let result = myRegex.test(myString); // bool returns as true due to built-in test() functionality
```

when the hell would you need this kind of warped functionality? password validation, email format validation, string reformatting, updating legacy code (eg changing function notation to arrow notation in a codebase).

I'm not going to do these challenges, because the use cases of regex are limited to authentication/validation, and large-scale software engineering, whereas I'm more interesting in web development as a means to prototype ideas, not to execute at scale (though I will look more deeply into software engineering best practices towards the end of the year).

debugging JavaScript

there are 1) syntax errors (IDE or terminal will catch these) 2) runtime errors (some loop is structured so it runs indefinitely etc.) 3) logical errors (program is not doing what you intended), and most of the time developers spend debugging, it's addressing problems 2 and 3.

use *typeof(),* an innate jS function to check the type of some stored variable/value. these typeof errors are particularly common when working with external JSON data, where a certain key you thought was an int was actually a string, or something along those lines.

be careful with inclusion of brackets when calling a function—
var = function just means var is now a function identical to *function*
var = function() means that var is equal to the output of function()—the return value, that is

be careful when calling functions from a library or someone else's code that you input the arguments in the correct order, lest you run into a logical error

use caution when re-initialising variables inside a loop, because you often get errors in programs with loops, or nested loops, because a particular variable isn't being re-initilialised, or something is wrong with it's scoping (see let vs var).

data structures (arrays and objects)

splice() is used to remove arbitrary elements from any position within an array, taking 3 arguments, where the first two are which location you want to start at, and then how many elements you want to remove from the array. this function, of course, is innate, and returns the values that were spliced from the array, much like pop() returns the value it removed from the end of the array. the third argument it takes is the elements you want to add to the array, starting from the index you gave earlier as a parameter. thus the function is very useful for switching elements out/swapping values in an array (variables can be passed as arguments also).

slice() does the same as splice() but doesn't modify the array, only copies the value we're interested in.

you can combine arrays using the spread() operator we learned about earlier (…arr2) by simple inserting …arr2 into the position you want arr2 to be inserted in a completely different array.

arr.indexOf(element) tells you the index of a particular element in the array, returning -1 if the element isn't in the array

there are lots of native functions built to handle array elements individually, including every(), forEach() and map(), but the simple for loop is in many situations the simplest way to manipulate arrays.

as well as object.hasOwnProperty(propertyName), you can user propertyName *in* object to get back a bool representing whether an object has a certain property.

Object.keys(nameOfObject) is what you use to return an array of all the properties that the object you passed in had.

the replace() method can be useful to query for some value in a string and replace it with some other value

whereas normally you'd have to set var = var.split(), you don't have to do that with splice(), which actually changes the variables naturally (slice does the same without actually changing it).

a property of a jS object defines its behaviour. you define a method within an object like any other property:

method: function() { // code ;}

you can create a jS object directly, by literally defining it and it's properties, or you can define a constructor function to abstract the creation process away so you can create multiple instances of that class (many objects) by simply calling in the constructor function and passing in the property values you want. the constructor function is the rough equivalent of python classes, which are general versions of python objects.

```
function Bird(name, wings) {
	this.name = 'someName';
	this.wings = 2;
}
```

// this is an example of a constructor function in use. when you call it to "instantiate" an object, you do blueBird = new Bird and the ensuing object is an INSTANCE of that constructor. you can check whether an object was created using a constructor, or independently via definition, using object instanceOf(constructor)

you can use hasOwnProperty() to check if an instance has a certain property, or even for more subtle manipulations like adding all the properties of a certain once to an array, etc.

you can use constructor.prototype.property to set a default property for all instances of the constructor (all child objects) so that you don't waste memory setting up 100 versions of an object that all have numLegs = 2;

extending this further, you can create a new "object" that just contains the information about the prototypes, i.e. the values that all the instances of the constructor will have by default, like this:

```
Bird.prototype = {
        constructor: Bird;
        numLegs = 2;
        eat: function() {
        return "nom nom";
        }
}
```
and this way you have numLegs and eat() ready to go in all instances of the Bird constructor function (and when you create a new object you can manipulate blueBird = new Bird, blueBird.name = and all that anyways, but these are the properties you don't plan on manipulating in the first place.

in the same vein that you have "isInstanceOf()" you have isPrototypeOf()

a prototype is just the general version of. so Dog is the object that's the general version of "James" and Object is the general version of Dog, and so this is called a prototype chain as you go up levels of abstraction.

if you find that your constructors are repeating things (eg Bear and Cat both have the same "name" property of just returning the name) you can create a new constructor that abstracts that away called, say, Animal, so you don't have to repeat yourself. in this sense, Animal is a "supertype" of Bear and Cat, since it's a layer of abstraction above.

everything in jS is an object—when you declare an array, that's an object, and jS has native prototypes that declare all the methods that can be used by instances of the Array object, which is how you can so easily just use .length, for example

a prototype is just another *property of the constructor function*, albeit one that encapsulates several methods and commonalities that every instance of the constructor should have built into it (like numLegs = 2)

jS doesn't have classes like other languages, the equivalent, instead, would be the constructor function, which is how you instantiate objects.

you can also have a new constructor that you create inherit properties and behaviours from some other constructor. it'd inherit properties using the ConstructorToInheritFrom.apply(this, argsConstructorTakes) and it'd inherit methods using prototype inheritance from the supertype, done via: newCon.prototype = Object.create(ConstructorToInheritFrom.prototype) since prototypes contain methods/behaviour

// you extend an inherited prototype and add new functionality just by doing newCon.prototype.newFunc = function() {};

so, for prototypical inheritance from Animal to Dog to instance(Beagle), once Dog has been declared as a constructor:
Dog.prototype = Object.create(Animal.prototype)l
let beagle = new Dog(); // instantiates so now beagle has animal behaviour

when this inheritance occurs, beagle would also register a constructor of Animal, when really you want a constructors of Dog, and so you have to change Dog.prototype.constructor = Dog before you let beagle = new Dog();

you'd use "mixins" to give common properties to unrelated objects. say you had a baby and football player, both of them will "kick" (one in the mother's womb, another a ball around a field) but inheritance doesn't make sense here since they aren't related. you'd create a "mixin" function via:

```
kickMixin = function(obj) {
        obj.kick = function() {
                return "SMACK!";
        };

};

kickMixin(baby);
kickMixin(footballPlayer);
```

then both baby.kick and fbp.kick would exist, despite the two still having no relationship to each other.

you can create "closure" to hide properties of objects by creating get() and set() functions (usually not the latter since the whole point of closing off a property of an object is that it can be read but not written to). this is useful if you're building a fintech startup mvp that deals with sensitive information using node.js or if your front end ever contains sensitive information in the form of constructed instances.

you can immediately call anonymous functions by doing (function() {})(); and this is called using an immediately invoked function expression (IIFE). an example use case is in the creation of modules, which package together related functions. say you declared a few different mixin functions and packaged them up into a module, you want to be able to call module.mixinName(object) to imbue that object with that behaviour at any time, but you can't if the functions haven't been "returned" and so you implement an IIFE to return the functions for use when you call the mixinName from the module. IIFEs are often used with closures to hide write access to certain internal value within modules while permitting access to read access in those modules.

functional programming

the "state" of an application (common notion in React) is just defined by the value of the variables, objects, and data inside the program.

functional programming is all about mathematical functions as small pieces that comprise a computer program. the functionality of each of the pieces, on its own, is independent of the state of the program (value of global variables, for example). these functions are all deterministic. the main difference is that functional programming is stateless, whereas OOP is dependent on system states. jS is not OOP language nor functional—it is a procedural programming language.

OOP contains data and functions within objects, and you get the objects to interact to achieve some end behaviour or computation. in functional programming, functions can be treated as variables, passed as arguments, and are pure (deterministic).

very few languages are only capable of one paradigm or the other. each paradigm is just a preference, where one may be better for solving a particular class of problem than another. javascript can lend itself to functional programming OR oop, whereas other languages may lend themselves more easily to other paradigms.

Lambda calculus has to do with functional programming.

pure functions are important in functional programming because we're looking to minimise change in state. so your a, b variables shouldn't have their values changed by being passed into a function (that's OOP), merely computed on to produce some clean output. thus, pure functions produce very few "side effects" which makes debugging that much easier.

a few examples will clear things up. functional programming is just a different, more abstract and cleaner way of programming, compared to procedural or object oriented. all of these methods discuss how to write code to achieve the same end result (ultimately the same task is being executed).

if you had:

procedural programming:

```
var L = [1, 2, 3]; // declare a list, say you wanted to increase each element by 1, you might:
for (let i = 0; i < L.length; i++) {
        L[i]++;
};
console.log(L); // prints [2, 3, 4]
```

functional programming:

```
var L = [1, 2, 3];
increment = (element) => element + 1;
return L.map(increment);
```

// notice how you're passing a function into a function, with each function doing one isolated task on the data
// notice how the code is much shorter and cleaner
// notice how you're not implementing the iteration over the array elements yourself (done under the hood in the map() method) and therefore are thinking at a higher, more abstract level
// functional programming will often just have many stacked/nested functions that do an operation then pass it to the next function, without worrying how the operation is actually implemented

a callback function is a function that takes in another function as an argument. the reason it gets is name is because the function will be invoked/execute when the first function returns some value. in other words, it is *called back*.

imperative programming (procedural is how we normally program, a subset of imperative) we give the computer commands to change the state of a system. declarative programming is where you declare *what you want done—the end result*, like "I want all elements in the array incremented by 1" and then the map() function will take care of the implementation for you.

essentially, the problem with imperative programming is that you make changes to state, and so it's very easy to screw an object up and not know where your data is getting screwed up inadvertently. Whereas with functional programming (and declarative programming more broadly), your functions are pure and have no side effects—they are not mutating state, and so it's much cleaner, more abstract, and easier to see where problems are occurring. for example, in the fCC tabs program, using splice() was actually changing the tabs in existence, so the function afterwards that wanted to grab the tabs after the index of the one you wanted to remove became confusing to implement, because the process of storing the tabs before the one you want to remove mutated the thing hat the second function was going to act on.

another principle of functional programming in general is to “declare dependencies explicitly”. what that means is if a function depends on a certain global variable, that var should be passed in as an argument. this makes things further easier to debug because you know exactly what values could be going wrong if a function depends on outside values.

when you’re dealing with global variables and writing functional code, just make copies of those and operate on those, if necessary to make manipulations.

map() and filter() useful to functional programmers because it doesn’t mutate the object passed in as an argument.

.slice() also does not modify its input and is therefore also useful to the functional programmer, and it takes in the position to begin slice, and end slice, non-inclusive

.concat() is another one joining two strings or arrays together and returning the combination, without actually mutating either of the original arrays

you can implement all of the imperative functions like .push(), .splice() etc in a functional manner using the functions above.

.reduce() is the most general operation that DOES mutate the input (and is thus not used in functional programming very often) and .map() and .filter() can be derived from it (are special cases). it combines several array elements into one, extracting data from the array—like taking an average.

you need to provide a callback function for the .sort() method to tell it how exactly to sort your elements, since it only does ascending and alphabetical by default. the callback function is usually called the “compare function” since it outlines how the values will be compared.