Naomi's freeCodeCamp Review

Hello! This PDF contains all of the review pages (unless I missed one) from freeCodeCamp's full stack developer curriculum.

Before we dive in, a quick disclaimer: This is not an officially sanctioned document, and I make no warranty that the information in this document will be kept up to date. This version was created on 7 October 2025.

Questions? Comments? Document is out of date and you want to scream at me about it? https://chat.nhcarrigan.com

HERE WE GO!

Asynchronous JavaScript Review

- **Synchronous JavaScript** is executed sequentially and waits for the previous operation to finish before moving on to the next one.
- **Asynchronous JavaScript** allows multiple operations to be executed in the background without blocking the main thread.
- **Thread** is a sequence of instructions that can be executed independently of the main program flow.
- **Callback functions** are functions that are passed as arguments to other functions and are executed after the completion of the operation or as a result of an event.

The JavaScript engine and JavaScript runtime

- The **JavaScript engine** is a program that executes JavaScript code in a web browser. It works like a converter that takes your code, turns it into instructions that the computer can understand and work accordingly.
- V8 is an example of a JavaScript engine developed by Google.
- The **JavaScript runtime** is the environment in which JavaScript code is executed. It includes the JavaScript engine which processes and executes the code, and additional features like a web browser or Node.js.

The Fetch API

- The Fetch API allows web apps to make network requests, typically to retrieve or send data to the server. It provides a `fetch()` method that you can use to make these requests.
- You can retrieve text, images, audio, JSON, and other types of data using the Fetch API.

HTTP methods for Fetch API

The Fetch API supports various HTTP methods to interact with the server. The most common methods are:

- **GET**: Used to retrieve data from the server. By default, the Fetch API uses the `GET` method to retrieve data.

```
fetch('https://api.example.com/data')
```

To use the fetched data, it must be converted to JSON format using the `.json()` method:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
```

In this code, the response coming from the Fetch API is a promise and the `.then` handler is converting the response to a JSON format.

- **POST**: Used to send data to the server. The `POST` method is used to create new resources on the server.

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
```

In this example, we're sending a `POST` request to create a new user. We have specified the method as `POST`, set the appropriate headers, and included a body with the data we want to send. The body needs to be a string, so we use `JSON.stringify()` to convert our object to a JSON string.

- **PUT**: Used to update data on the server. The `PUT` method is used to update existing resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Smith',
    email: 'john@example.com'
  })
})
```

In this example, we are updating the ID `45` that is specified at the end of the URL. We have used the `PUT` method on the code and also specified the data as the body which will be used to update the identified data.

- **DELETE**: Used to delete data on the server. The `DELETE` method is used to delete resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'DELETE'
})
```

In this example, we're sending a `DELETE` request to remove a user with the ID `45`.

Promise and promise chaining

- **Promises** are objects that represent the eventual completion or failure of an asynchronous operation and its resulting value. The value of the promise is known only when the `async` operation is completed.
- Here is an example to create a simple promise:

```
```

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data received successfully');
  }, 2000);
});
```

- The `.then()` method is used in a Promise to specify what should happen when the Promise is fulfilled, while `.catch()` is used to handle any errors that occur.
- Here is an example of using `.then()` and `.catch()` with a Promise:

```
promise
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, the `.then()` method is used to log the data received from the Promise, while the `.catch()` method is used to log any errors that occur.

- **Promise chaining**: One of the powerful features of Promises is that we can chain multiple asynchronous operations together. Each `.then()` can return a new Promise, allowing you to perform a sequence of asynchronous operations one after the other.
- Here is an example of Promise chaining:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return fetch('https://api.example.com/other-data');
  })
  .then(response => response.json())
  .then(otherData => {
    console.log(otherData);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, we first fetch data from one URL, then fetch data from another URL based on the first response, and finally log the second data received.

The `catch` method would handle any errors that occur during the process. This means you don't need to add error handling to each step, which can greatly simplify your code.

Using `async/await` to handle promises

Async/await makes writing & reading asynchronous code easier which is built on top of Promises.

- **async**: The `async` keyword is used to define an asynchronous function. An `async` function returns a Promise, which resolves with the value returned by the `async` function.
- **await**: The `await` keyword is used inside an `async` function to pause the execution of the function until the Promise is resolved. It can only be used inside an `async` function.
- Here is an example of using `async/await`:

```
```

```
async function delayedGreeting(name) {
  console.log("A Messenger entered the chat...");
  await new Promise(resolve => setTimeout(resolve, 2000));
  console.log(`Hello, ${name}!`);
}

delayedGreeting("Alice");
console.log("First Printed Message!");
```

In the above example, the `delayedGreeting` function is an `async` function that pauses for 2 seconds before printing the greeting message. The `await` keyword is used to pause the function execution until the `Promise` is resolved.

- One of the biggest advantages of `async/await` is error handling via `try/catch` blocks. Here's an example:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

fetchData();
```

In the above example, the `try` block contains the code that might throw an error, and the `catch` block handles the error if it occurs. This makes error handling more straightforward and readable.

The `async` attribute

- The `async` attribute tells the browser to download the script file asynchronously while continuing to parse the HTML document.
- Once the script is downloaded, the HTML parsing is paused, the script is executed, and then HTML parsing resumes.
- You should use `async` for independent scripts where the order of execution doesn't matter

The `defer` attribute

- The `defer` attribute also downloads the script asynchronously, but it defers the execution of the script until after the HTML document has been fully parsed.
- The `defer` scripts maintain the order of execution as they appear in the HTML document.
- It's important to note that both `async` and `defer` attributes are ignored for inline scripts and only work for external script files.
- When both `async` and `defer` attributes are present, the `async` attribute takes precedence.

Geolocation API

- The Geolocation API provides a way for websites to request the user's location.
- The example below demonstrates the API's `getCurrentPosition()` method which is used to get the user's current location.

```
```

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log("Latitude: " + position.coords.latitude);
    console.log("Longitude: " + position.coords.longitude);
  },
  (error) => {
    console.log("Error: " + error.message);
  }
);
```

In this code, we're calling `getCurrentPosition` and passing it a function which will be called when the position is successfully obtained.

The `position` object contains a variety of information, but here we have selected `latitude` and `longitude` only.

If there is an issue with getting the `position`, then the error will be logged to the console.

- It is important to respect the user's privacy and only request their location when necessary.

--assignment--

Review the Asynchronous JavaScript topics and concepts.

Bash and SQL Review

Database Normalization

This is the process of organizing a relational database to reduce data redundancy and improve integrity.

Its benefits include:

- Minimizing duplicated data, which saves storage and reduces inconsistencies.
- Enforcing data integrity through the use of primary and foreign keys.
- Making databases easier to maintain and understand.

Normal Forms

- **1NF (First Normal Form)**
  - Each cell contains a single (atomic) value.
  - Each record is unique (enforced by a primary key).
  - Order of rows/columns is irrelevant.
  - Example: Move multiple phone numbers from a `students` table into a separate `student_phones` table.
- **2NF (Second Normal Form)**
  - Meets 1NF requirements.
  - No **partial dependencies**: every non-key attribute must depend on the entire composite primary key.
  - Example: Split `orders` table into `order_header` and `order_items` to avoid attributes depending on only part of the key.
- **3NF (Third Normal Form)**
  - Meets 2NF requirements.
  - No **transitive dependencies**: non-key attributes cannot depend on other non-key attributes.

    - Example: Move `city_postal_code` to a `cities` table instead of storing it with every order.

- **BCNF (Boyce-Codd Normal Form)**
    - Meets 3NF requirements.
    - Every determinant (left-hand side of a functional dependency) must be a superkey.

Tip: Aim for 3NF in most designs for a good balance of integrity and performance.

Key SQL Concepts

- SQL is a Structured Query Language for communicating with relational databases.
- **Basic commands** → `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, `ALTER TABLE`, etc.
- `Joins` → Combines data from multiple tables (`INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`).

Running SQL Commands in Bash

You can run SQL commands directly from the command line using the `psql` command-line client for PostgreSQL or similar tools for other databases.

For example, to run a SQL file in PostgreSQL:

```
psql -U username -d database_name -c "SELECT * FROM students;"
```

You can also execute MySQL commands directly:

```
mysql -u username -p database_name -e "SELECT * FROM students;"
```

Run SQL from a File

```
# PostgreSQL
psql -U username -d database_name -f script.sql

# MySQL
mysql -u username -p database_name < script.sql
```

Embed SQL in a Bash Script

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Insert student data
psql -U "$DB_USER" -d "$DB_NAME" -c \
"INSERT INTO students (name, age, major) VALUES ('Alice', 20, 'CS');"
```

Use of Variables in SQL

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"
STUDENT_NAME="Bob"
AGE=21
```

```
psql -U "$DB_USER" -d "$DB_NAME" -c \
"INSERT INTO students (name, age) VALUES ('$STUDENT_NAME', $AGE);"
```

Tip: Sanitize variables to avoid SQL injection.

Retrieving and Using SQL Query Results in Bash

When you run SQL queries via `psql`, you can **capture** and **process** the returned values in your Bash scripts.

Capturing a Single Value

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get total student count
STUDENT_COUNT=$(psql -U "$DB_USER" -d "$DB_NAME" -t -A -c \
"SELECT COUNT(*) FROM students;")

echo "Total students: $STUDENT_COUNT"
```

Output → 42

Retrieving Multiple Columns

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get top 3 students' names and ages
RESULTS=$(psql -U "$DB_USER" -d "$DB_NAME" -t -A -F"," -c \
"SELECT name, age FROM students LIMIT 3;")

echo "Top 3 students:"
echo "$RESULTS"
```

Output

```
Alice,20
Bob,21
Charlie,22
```

Looping Through Query Results

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get student names and majors
psql -U "$DB_USER" -d "$DB_NAME" -t -A -F"," -c \
"SELECT name, major FROM students;" | while IFS="," read -r name major
do
    echo "Student: $name | Major: $major"
done
```

Shape of Output

```
Student: Alice | Major: CS
Student: Bob   | Major: Math
Student: Carol | Major: Physics
```

SQL Injection

It is a web security vulnerability where attackers insert malicious SQL code into input fields to manipulate the database.

This can lead to risky actions like:

- Bypassing authentication.
- Stealing sensitive data.
- Modifying or deleting records.

An example of an SQL injection attack:

```
SELECT * FROM users WHERE username = ' " " OR "1"="1" -- ' AND password =
'anything';
```

This query would return all users because the condition `OR "1"="1"` is always true, allowing attackers to bypass login checks.

Preventing SQL Injection

1. **Use Prepared Statements**: These separate SQL code from data, preventing injection. Here's an example (Node.js with pg):

   ```
   client.query('SELECT * FROM users WHERE username = $1 AND password = $2',
   [username, password]);
   ```

2. **Input Validation**: Sanitize and validate all user inputs to ensure they conform to expected formats.

3. **Least Privilege**: Use database accounts with the minimum permissions necessary for the application.

Note: Never grant admin rights to application accounts.

N+1 Problem

The N+1 problem occurs when an application makes one query to retrieve a list of items (N) and then makes an additional query for each item to retrieve related data, resulting in N+1 queries.

Why It's Bad

- Each query adds network and processing overhead.
- Multiple small queries are slower than one optimized query.

Example of N+1 Pattern

```
-- 1: Get list of orders
SELECT * FROM orders LIMIT 50;
```

```
-- N: For each order, get customer
SELECT * FROM customers WHERE customer_id = ...;
```

Solution: Use `JOINs` or other set-based operations.

```
SELECT
  orders.order_id,
  orders.product,
  orders.quantity,
  customers.customer_id,
  customers.name,
  customers.email,
  customers.address
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id
WHERE orders.order_id IN (SELECT order_id FROM orders LIMIT 50);
```

Always look for opportunities to combine related data into a single query.

--assignment--

Review the Bash and SQL topics and concepts.

Bash Commands Review

Terminal, Shell, and Command Line Basics

- **Command line**: A text interface where users type commands.
- **Terminal**: The application that provides access to the command line.
- **Terminal emulator**: Adds extra features to a terminal.
- **Shell**: Interprets the commands entered into the terminal (e.g., Bash).
- **PowerShell / Command Prompt / Microsoft Terminal**: Options for accessing the command line on Windows.
- **Terminal (macOS)**: Built-in option on macOS, with third-party alternatives like iTerm or Ghostty.
- **Terminal (Linux)**: Options vary by distribution, with many third-party emulators like kitty.
- **Terminology**: Though "terminal," "shell," and "command line" are often used interchangeably, they have specific meanings.

Command Line Shortcuts

- **Up/Down arrows**: Cycle through previous/next commands in history.
- **Tab**: Autocomplete commands.
- `Control+L` (Linux/macOS) or typing `cls` (Windows): Clear the terminal screen.
- `Control+C`: Interrupt a running command (also used for copying in PowerShell if text is selected).
- `Control+Z` (Linux/macOS only): Suspend a task to the background; use `fg` to resume it.
- `!!`: Instantly rerun the last executed command.

Bash Basics

- **Bash** (Bourne Again Shell): Widely used Unix-like shell.
  Key commands:
  - `pwd`: Show the current directory.
  - `cd`: Change directories.

        - `..` refers to the parent directory (one level up).
        - `.` refers to the current directory.
    - `ls`: List files and folders.
        - `-a`: Show all files, including hidden files.
        - `-l`: Show detailed information about files.
    - `less`: View file contents one page at a time with navigation options, including scrolling backward and searching.
    - `more`: Display file contents one screen at a time, with limited backward scrolling and basic navigation.
    - `cat`: Show the entire file content at once without scrolling or navigation, useful for smaller files.
    - `mkdir`: Create a new directory.
    - `rmdir`: Remove an empty directory.
    - `touch`: Create a new file.
    - `mv`: Move or rename files.
        - Rename: `mv oldname.txt newname.txt`
        - Move: `mv filename.txt /path/to/target/`
    - `cp`: Copy files.
        - `-r`: Recursively copy directories and their contents.
    - `rm`: Delete files.
        - `-r`: Recursively delete directories and their contents.
    - `echo`: Display a line of text or a variable's value.
        - Use `>` to overwrite the existing content in a file. (e.g., `echo "text" > file.txt`)
        - Use `>>` to append output to a file **without overwriting existing content** (e.g., `echo "text" >> file.txt`).
    - `exit`: Exit the terminal session.
    - `clear`: Clear the terminal screen.
    - `find`: Search for files and directories.
        - `-name`: Search for files by name pattern (e.g., `find . -name "*.txt"`).
    - Use `man` followed by a command (e.g., `man ls`) to access detailed manual/help pages.

Command Options and Flags

- **Options** or **flags**: modify a command's behavior and are usually prefixed with hyphens:
    - **Long form (two hyphens)**:
        - Example: `--help`, `--version`
        - Values are attached using an equals sign, e.g., `--width=50`.
    - **Short form (one hyphen)**:
        - Example: `-a`, `-l`
        - Values are passed with a space, e.g., `-w 50`.
        - Multiple short options can be chained together, e.g., `ls -alh`.
- `--help`: You can always use a command with this flag to understand the available options for any command.

--assignment--

Review the Bash Commands topics and concepts.

Bash Scripting Review

Bash Scripting Basics

- **Bash scripting**: Writing a sequence of Bash commands in a file, which you can then execute with Bash to run the contents of the file.

- **Shebang**: The commented line at the beginning of a script (e.g., `#!/bin/bash`) that indicates what interpreter should be used for the script.

```
#!/bin/bash
```

- **Variable assignment**: Instantiate variables using the syntax `variable_name=value`.

```
servers=("prod" "dev")
```

- **Variable creation rules**: Create variables with `VARIABLE_NAME=VALUE` syntax. No spaces are allowed around the equal sign (`=`). Use double quotes if the value contains spaces.

```
NAME=John
MESSAGE="Hello World"
COUNT=5
TEXT="The next number is, "
```

- **Variable usage**: Access variable values by placing `$` in front of the variable name.

```
echo $NAME
echo "The message is: $MESSAGE"
```

- **Variable interpolation**: Use `$variable_name` to access the value of a variable within strings and commands.

```
TEXT="The next number is, "
NUMBER=42
echo $TEXT B:$NUMBER
echo $TEXT I:$NUMBER

echo "Pulling $server"
rsync --archive --verbose $server:/etc/nginx/conf.d/server.conf
configs/$server.conf
```

- **Variable scope**: Shell scripts run from top to bottom, so variables can only be used below where they are created.

```
NAME="Alice"
echo $NAME
```

- **User input**: Use `read` to accept input from users and store it in a variable.

```
read USERNAME
echo "Hello $USERNAME"
```

- **Comments**: Add comments to your scripts using `#` followed by your comment text.
  - Single-line comments start with `#` and continue to the end of the line
  - Comments are ignored by the shell and don't affect script execution

```
# This is a single-line comment
NAME="John"  # Comment at end of line
```

- **Multi-line comments**: Comment out blocks of code using colon and quotes.

```
: '
This is a multi-line comment
Everything between the quotes is ignored
Useful for debugging or documentation
'
```

- **Built-in commands and help**:
  - Use `help` to see a list of built-in bash commands
  - Use `help <command>` to get information about specific built-in commands
  - Some commands (like `if`) are built-ins and don't have man pages
  - Built-in commands are executed directly by the shell rather than as external programs
  - Use `help function` to see information about creating functions

```
help
help if
help function
```

- **Finding command locations**: Use `which` to locate where executables are installed.
  - Shows the full path to executable files
  - Useful for finding interpreter locations (like bash)
  - Helps verify which version of a command will be executed

```
which bash
which python
which ls
```

- **Manual pages**: Use `man` to access detailed documentation for commands.
  - Provides comprehensive information about command usage
  - Shows all available options and examples
  - Use arrow keys to navigate, 'q' to quit
  - Not all commands have manual pages (built-ins use `help` instead)

```
man echo
man ls
man bash
```

- **Help flags**: Many commands support `--help` for quick help information.
  - Alternative to manual pages for quick reference
  - Shows command syntax and common options
  - Not all commands support this flag (some may show error)

```
ls --help
chmod --help
```

```
mv --help
```

- **Echo command options**: The `echo` command supports various options:
  - `-e` option enables interpretation of backslash escapes
  - `\n` creates a new line
  - Empty lines are only printed when values are enclosed in quotes
  - Useful for creating formatted output and program titles

```
echo -e "Line 1\nLine 2"
echo ""
echo -e "\n~~ Program Title ~~\n"
echo "Line 1\nLine 2"
```

- **Script arguments**: Programs can accept arguments that are accessible using `$` variables.
  - `$*` prints all arguments passed to the script
  - `$@` prints all arguments passed to the script as separate quoted strings
  - `$<number>` accesses specific arguments by position (e.g., `$1`, `$2`, `$3`)

```
echo $*
echo $@
echo $1
echo $2
```

Double Bracket Expressions `[[ ]]`

- **Double bracket syntax**: Use `[[ ]]` for conditional testing and pattern matching.
  - Must have spaces inside the brackets and around operators
  - Returns exit status 0 (true) or 1 (false) based on the test result

```
[[ $variable == "value" ]]
[[ $number -gt 10 ]]
[[ -f filename.txt ]]
```

- **String comparison operators**: Compare strings using various operators within `[[ ]]`.
  - `==` (equal): Tests if two strings are identical
  - `!=` (not equal): Tests if two strings are different
  - `<` (lexicographically less): String comparison in alphabetical order
  - `>` (lexicographically greater): String comparison in alphabetical order

```
[[ "apple" == "apple" ]]
[[ "apple" != "orange" ]]
[[ "apple" < "banana" ]]
[[ "zebra" > "apple" ]]
```

- **Numeric comparison operators**: Compare numbers using specific numeric operators.
  - `-eq` (equal): Numeric equality comparison
  - `-ne` (not equal): Numeric inequality comparison
  - `-lt` (less than): Numeric less than comparison
  - `-le` (less than or equal): Numeric less than or equal comparison

- `-gt` (greater than): Numeric greater than comparison
- `-ge` (greater than or equal): Numeric greater than or equal comparison

```
[[ $number -eq 5 ]]
[[ $count -ne 0 ]]
[[ $age -ge 18 ]]
[[ $score -lt 100 ]]
```

- **Logical operators**: Combine multiple conditions using logical operators.
  - `&&` (and): Both conditions must be true
  - `||` (or): At least one condition must be true
  - `!` (not): Negates the condition (makes true false, false true)

```
[[ $age -ge 18 && $age -le 65 ]]
[[ $name == "John" || $name == "Jane" ]]
[[ ! -f missing_file.txt ]]
```

- **File test operators**: Test file properties and existence.
  - `-e file`: True if file exists
  - `-f file`: True if file exists and is a regular file
  - `-d file`: True if file exists and is a directory
  - `-r file`: True if file exists and is readable
  - `-w file`: True if file exists and is writable
  - `-x file`: True if file exists and is executable
  - `-s file`: True if file exists and has size greater than zero

```
[[ -e /path/to/file ]]
[[ -f script.sh ]]
[[ -d /home/user ]]
[[ -x program ]]
```

- **Pattern matching with `=~`**: Use regular expressions for advanced pattern matching.
  - `=~` operator enables regex pattern matching
  - Pattern should not be quoted when using regex metacharacters
  - Supports full regular expression syntax
  - Case-sensitive by default

```
[[ "hello123" =~ [0-9]+ ]]
[[ "email@domain.com" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
]]
[[ "filename.txt" =~ \.txt$ ]]
```

- **Variable existence testing**: Check if variables are set or empty.
  - Test if variable is empty: `[[ ! $variable ]]`

```
[[ ! $undefined_var ]]
```

Double Parentheses Expressions `(( ))`

- **Arithmetic evaluation**: Use `(( ))` for mathematical calculations and numeric comparisons.
  - Evaluates arithmetic expressions using C-style syntax
  - Variables don't need `$` prefix inside double parentheses
  - Returns exit status 0 if result is non-zero, 1 if result is zero
  - Supports all standard arithmetic operators

```
(( result = 10 + 5 ))
(( count++ ))
(( total += value ))
```

- **Arithmetic operators**: Mathematical operators available in `(( ))`.
  - `+` (addition): Add two numbers
  - `-` (subtraction): Subtract second number from first
  - `*` (multiplication): Multiply two numbers
  - `/` (division): Divide first number by second (integer division)
  - `%` (modulus): Remainder after division
  - `**` (exponentiation): Raise first number to power of second

```
(( sum = a + b ))
(( diff = x - y ))
(( product = width * height ))
(( remainder = num % 10 ))
(( power = base ** exponent ))
```

- **Assignment operators**: Modify variables using arithmetic assignment operators.
  - `=` (assignment): Assign value to variable
  - `+=` (add and assign): Add value to variable
  - `-=` (subtract and assign): Subtract value from variable
  - `*=` (multiply and assign): Multiply variable by value
  - `/=` (divide and assign): Divide variable by value
  - `%=` (modulus and assign): Set variable to remainder

```
(( counter = 0 ))
(( counter += 5 ))
(( total -= cost ))
(( area *= 2 ))
(( value /= 3 ))
```

- **Increment and decrement operators**: Modify variables by one.
  - `++variable` (pre-increment): Increment before use
  - `variable++` (post-increment): Increment after use
  - `--variable` (pre-decrement): Decrement before use
  - `variable--` (post-decrement): Decrement after use

```
(( ++counter ))
(( index++ ))
(( --remaining ))
(( attempts-- ))
```

- **Comparison operators**: Compare numbers using arithmetic comparison.

- `==` (equal): Numbers are equal
- `!=` (not equal): Numbers are not equal
- `<` (less than): First number is less than second
- `<=` (less than or equal): First number is less than or equal to second
- `>` (greater than): First number is greater than second
- `>=` (greater than or equal): First number is greater than or equal to second

```
(( age >= 18 ))
(( score < 100 ))
(( count == 0 ))
(( temperature > freezing ))
```

- **Logical operators**: Combine arithmetic conditions.
  - `&&` (and): Both conditions must be true
  - `||` (or): At least one condition must be true
  - `!` (not): Negates the condition

```
(( age >= 18 && age <= 65 ))
(( score >= 90 || extra_credit > 0 ))
(( !(count == 0) ))
```

- **Bitwise operators**: Perform bit-level operations on integers.
  - `&` (bitwise AND): AND operation on each bit
  - `|` (bitwise OR): OR operation on each bit
  - `^` (bitwise XOR): XOR operation on each bit
  - `~` (bitwise NOT): Invert all bits
  - `<<` (left shift): Shift bits to the left
  - `>>` (right shift): Shift bits to the right

```
(( result = a & b ))
(( flags |= new_flag ))
(( shifted = value << 2 ))
```

- **Conditional (ternary) operator**: Use `condition ? true_value : false_value` syntax.
  - Provides a concise way to assign values based on conditions
  - Similar to the ternary operator in C-style languages
  - Evaluates condition and returns one of two values

```
(( result = (score >= 60) ? 1 : 0 ))
(( max = (a > b) ? a : b ))
(( sign = (num >= 0) ? 1 : -1 ))
```

- **Command substitution with arithmetic**: Use `$(( ))` to capture arithmetic results.
  - Returns the result of the arithmetic expression as a string
  - Can be used in assignments or command arguments
  - Useful for calculations that need to be used elsewhere

```
result=$(( 10 + 5 ))
echo "The answer is $(( a * b ))"
```

```
array_index=$(( RANDOM % array_length ))
```

Control Flow and Conditionals

- **Conditional statements**: Use `if` statements to execute code based on conditions.
  - Basic syntax: `if [[ CONDITION ]] then STATEMENTS fi`
  - Full syntax: `if [[ CONDITION ]] then STATEMENTS elif [[ CONDITION ]] then STATEMENTS else STATEMENTS fi`
  - Can use both `[[ ]]` and `(( ))` expressions for different types of conditions
  - **elif (else if)**: Optional, can be repeated multiple times to test additional conditions in sequence
  - **else**: Optional, executes when all previous conditions are false
  - Can mix double parentheses `(( ... ))` and double brackets `[[ ... ]]` in same conditional chain

```
if (( NUMBER <= 15 ))
then
    echo "B:$NUMBER"
elif [[ $NUMBER -le 30 ]]
then
    echo "I:$NUMBER"
elif (( NUMBER < 46 ))
then
    echo "N:$NUMBER"
elif [[ $NUMBER -lt 61 ]]
then
    echo "G:$NUMBER"
else
    echo "O:$NUMBER"
fi
```

Command Execution and Process Control

- **Command separation**: Use semicolon (`;`) to run multiple commands on a single line.
  - Commands execute sequentially from left to right
  - Each command's exit status can be checked individually

```
[[ 4 -ge 5 ]]; echo $?
ls -l; echo "Done"
```

- **Exit status**: Every command has an exit status that indicates success or failure.
  - Access exit status of the last command with `$?`
  - Exit status `0` means success (true/no errors)
  - Any non-zero exit status means failure (false/errors occurred)
  - Common error codes: `127` (command not found), `1` (general error)

```
echo $?
[[ 4 -le 5 ]]; echo $?
ls; echo $?
bad_command; echo $?
```

- **Subshells and command substitution**: Different uses of parentheses for execution contexts.

- Single parentheses `( ... )` create a subshell
- `$( ... )` performs command substitution
- Subshells run in separate environments and don't affect parent shell variables

```
( cd /tmp; echo "Current dir: $(pwd)" )
current_date=$(date)
file_count=$(ls | wc -l)
echo "Today is $current_date"
echo "Found $file_count files"
```

- **Sleep command**: Pause script execution for a specified number of seconds.
  - Useful for creating delays in scripts
  - Can be used with decimal values for subsecond delays

```
sleep 3
sleep 0.5
sleep 1
```

Loops

- **While loops**: Execute code repeatedly while a condition is true.
  - Syntax: `while [[ CONDITION ]] do STATEMENTS done`

```
I=5
while [[ $I -ge 0 ]]
do
    echo $I
    (( I-- ))
    sleep 1
done
```

- **Until loops**: Execute code repeatedly until a condition becomes true.
  - Syntax: `until [[ CONDITION ]] do STATEMENTS done`

```
until [[ $QUESTION =~ \?$ ]]
do
    echo "Please enter a question ending with ?"
    read QUESTION
done
until [[ $QUESTION =~ \?$ ]]
do
    GET_FORTUNE again
done
```

- **For loops**: Iterate through arrays or lists using `for` loops with `do` and `done` to define the loop's logical block.

```
for server in "${servers[@]}"
do
    echo "Processing $server"
done
for (( i = 1; i <= 5; i++ ))
```

```
do
    echo "Number: $i"
done
for (( i = 5; i >= 1; i-- ))
do
    echo "Countdown: $i"
done
for i in {1..5}
do
    echo "Count: $i"
done
```

Arrays

- **Arrays**: Store multiple values in a single variable.
  - Create arrays with parentheses: `ARRAY=("value1" "value2" "value3")`
  - Access elements by index: `${ARRAY[0]}`, `${ARRAY[1]}`
  - Access all elements: `${ARRAY[@]}` or `${ARRAY[*]}`
  - Array indexing starts at 0

```
RESPONSES=("Yes" "No" "Maybe" "Ask again later")
echo ${RESPONSES[0]}     # Yes
echo ${RESPONSES[1]}     # No
echo ${RESPONSES[5]}     # Index 5 doesn't exist; empty string
echo ${RESPONSES[@]}     # Yes No Maybe Ask again later
echo ${RESPONSES[*]}     # Yes No Maybe Ask again later
```

- **Array inspection with declare**: Use `declare -p` to view array details.
  - Shows the array type with `-a` flag
  - Displays all array elements and their structure

```
ARR=("a" "b" "c")
declare -p ARR # ARR=([0]="a" [1]="b" [2]="c")
```

- **Array expansion**: Use `"${array_name[@]}"` syntax to expand an array into individual elements.

```
for server in "${servers[@]}"
```

Functions

- **Functions**: Create reusable blocks of code.
  - Define with `FUNCTION_NAME() { STATEMENTS }`
  - Call by using the function name
  - Can accept arguments accessible as `$1`, `$2`, etc.

```
GET_FORTUNE() {
    echo "Ask a question:"
    read QUESTION
}
GET_FORTUNE
```

- **Function arguments**: Functions can accept arguments just like scripts.
  - Arguments are passed when calling the function
  - Access arguments inside function using `$1`, `$2`, etc.
  - Use conditional logic to handle different arguments

```
GET_FORTUNE() {
    if [[ ! $1 ]]
    then
        echo "Ask a yes or no question:"
    else
        echo "Try again. Make sure it ends with a question mark:"
    fi
    read QUESTION
}
GET_FORTUNE
GET_FORTUNE again
```

Random Numbers and Mathematical Operations

- **Random numbers**: Generate random values using the `$RANDOM` variable.
  - `$RANDOM` generates numbers between 0 and 32767
  - Use modulus operator to limit range: `$RANDOM % 75`
  - Add 1 to avoid zero: `$(( RANDOM % 75 + 1 ))`
  - Must use `$(( ... ))` syntax for calculations with `$RANDOM`

```
NUMBER=$(( RANDOM % 6 ))
DICE=$(( RANDOM % 6 + 1 ))
BINGO=$(( RANDOM % 75 + 1 ))
echo $(( RANDOM % 10 ))
```

- **Random array access**: Use random numbers to access array elements randomly.
  - Generate random index within array bounds
  - Use random index to access array elements
  - Useful for random selections from predefined options

```
RESPONSES=("Yes" "No" "Maybe" "Outlook good" "Don't count on it" "Ask
again later")
N=$(( RANDOM % 6 ))
echo ${RESPONSES[$N]}
```

- **Modulus operator**: Use `%` to get the remainder of division operations.
  - Essential for limiting random number ranges
  - Works with `$RANDOM` to create bounded random values
  - `RANDOM % n` gives numbers from 0 to n-1

```
echo $(( 15 % 4 ))
echo $(( RANDOM % 100 ))
echo $(( RANDOM % 10 + 1 ))
```

Environment and System Information

- **Environment variables**: Predefined variables available in the shell environment.
  - `$RANDOM`: Generates random numbers between 0 and 32767
  - `$LANG`: System language setting
  - `$HOME`: User's home directory path
  - `$PATH`: Directories searched for executable commands
  - View all with `printenv` or `declare -p`

```
echo $RANDOM
echo $HOME
echo $LANG
printenv
```

- **Variable inspection**: Use `declare` to view and work with variables.
  - `declare -p`: Print all variables and their values
  - `declare -p VARIABLE`: Print specific variable details
  - Shows variable type (string, array, etc.) and attributes

```
declare -p
declare -p RANDOM
declare -p MY_ARRAY
```

- **Command types**: Different categories of commands available in bash.
  - **Built-in commands**: Executed directly by the shell (e.g., `echo`, `read`, `if`)
  - **External commands**: Binary files in system directories (e.g., `ls`, `sleep`, `bash`)
  - **Shell keywords**: Language constructs (e.g., `then`, `do`, `done`)
  - Use `type <command>` to see what type a command is

```
type echo
type ls
type if
type ./script.sh
```

File Creation and Management

- **File creation**: Use `touch` to create new empty files.
  - Creates a new file if it doesn't exist
  - Updates the timestamp if the file already exists
  - Commonly used to create script files before editing

```
touch script.sh
touch bingo.sh
touch filename.txt
```

Creating and Running Bash Scripts

- **Script execution methods**: Multiple ways to run bash scripts:
  - **`sh scriptname.sh`**: Run with the sh shell interpreter.
  - **`bash scriptname.sh`**: Run with the bash shell interpreter.
  - **`./scriptname.sh`**: Execute directly (requires executable permissions).

```
sh questionnaire.sh
bash questionnaire.sh
./questionnaire.sh
```

File Permissions and Script Execution

- **Permission denied error**: When using `./scriptname.sh`, you may get "permission denied" if the file lacks executable permissions.
- **Checking permissions**: Use `ls -l` to view file permissions.

  ```
  ls -l questionnaire.sh
  ```

- **Permission format**: The output shows permissions as `-rw-r--r--` where:
  - First character (`-`): File type (- for regular file, d for directory)
  - Next 9 characters: Permissions for owner, group, and others
  - `r` = read, `w` = write, `x` = execute
- **Adding executable permissions**: Use `chmod +x` to give executable permissions to everyone.

  ```
  chmod +x questionnaire.sh
  ```

- **Script organization**: Best practices for structuring bash scripts.
  - Start with shebang (`#!/bin/bash`)
  - Add descriptive comments about script purpose
  - Define variables at the top
  - Group related functions together
  - Main script logic at the bottom

  ```
  #!/bin/bash
  NAME="value"
  ARRAY=("item1" "item2")
  my_function() {
      echo "Function code here"
  }
  my_function
  echo "Script complete"
  ```

- **Sequential script execution**: Create master scripts that run multiple programs in sequence.
  - Useful for automating workflows that involve multiple scripts
  - Each script runs to completion before the next one starts
  - Can combine different programs into a single execution flow
  - Arguments can be passed to individual scripts as needed
  - Can include different types of programs (interactive, automated, etc.)

  ```
  #!/bin/bash
  ./setup.sh
  ./interactive.sh
  ./processing.sh
  ./cleanup.sh
  ```

--assignment--

Review the Bash Scripting topics and concepts.

Basic CSS Review

CSS Basics

- **What is CSS?**: Cascading Style Sheets (CSS) is a markup language used to apply styles to HTML elements. CSS is used for colors, background images, layouts and more.
- **Basic Anatomy of a CSS Rule**: A CSS rule is made up of two main parts: a selector and a declaration block. A selector is a pattern used in CSS to identify and target specific HTML elements for styling. A declaration block applies a set of styles for a given selector or selectors.

Here is the general syntax of a CSS rule:

```
selector {
    property: value;
}
```

- `meta name="viewport"` **Element**: This `meta` element gives the browser instructions on how to control the page's dimensions and scaling on different devices, particularly on mobile phones and tablets.
- **Default Browser Styles**: Each HTML element will have default browser styles applied to them. This usually includes items like default margins and paddings.

Inline, Internal, and External CSS

- **Inline CSS**: These styles are written directly within an HTML element using the `style` attribute. Most of the time you will not be using inline CSS due to separation of concerns.

Here is an example of inline CSS:

```
<p style="color: red;">This is a red paragraph.</p>
```

- **Internal CSS**: These styles are written within the `<style>` tags inside the `head` section of an HTML document. This can be useful for creating short code examples, but usually you will not need be using internal CSS.
- **External CSS**: These styles are written in a separate CSS file and linked to the HTML document using the `link` element in the `head` section. For most projects, you will use an external CSS file over internal or inline CSS.

Working With the `width` and `height` Properties

- `width` **Property**: This property specifies the width of an element. If you do not specify a width, then the default is set to `auto`. This means the element will take up the full width of its parent container.
- `min-width` **Property**: This property specifies the minimum width for an element.
- `max-width` **Property**: This property specifies the maximum width for an element.
- `height` **Property**: This property specifies the height of an element. Similarly, the height is `auto` by default, which means it will adjust to the content inside.
- `min-height` **Property**: This property specifies the minimum height for an element.
- `max-height` **Property**: This property specifies the maximum height for an element.

Different Types of CSS Combinators

- **Descendant Combinator**: This combinator is used to target elements that are descendants of a specified parent element. The following example will target all `li` items inside `ul` elements.

```
<ul>
    <li>Example item one</li>
    <li>Example item two</li>
    <li>Example item three</li>
</ul>
```

```
ul li {
    background-color: yellow;
}
```

- **Child Combinator (`>`)**: This combinator is used to select elements that are direct children of a specified parent element. The following example will target all `p` elements that are direct children of the `container` class.

```
<div class="container">
  <p>This will get styled.</p>

  <div>
    <p>This will not get styled.</p>
  </div>
</div>
```

```
.container > p {
  background-color: black;
  color: white;
}
```

- **Next-sibling Combinator (`+`)**: This combinator selects an element that immediately follows a specified sibling element. The following example will select the paragraph element that immediately follows the `h2` element.

```
<h2>I am a sub heading</h2>

<p>This paragraph element will get a red background.</p>
```

```
h2 + p {
  background-color: red;
}
```

- **Subsequent-sibling Combinator (`~`)**: This combinator selects all siblings of a specified element that come after it. The following example will style only the second paragraph element because it is the only one that is a sibling of the `ul` element and shares the same parent.

```
<div class="container">
  <p>This will not get styled.</p>
  <ul>
    <li>Example item one</li>
```

```
    <li>Example item two</li>
    <li>Example item three</li>
  </ul>
  <p>This will get styled.</p>
</div>
<p>This will not get styled.</p>
```

```
ul ~ p {
  background-color: green;
}
```

Inline, Block, and Inline-Block Level Elements

- **Inline Level Elements**: Inline elements only take up as much width as they need and do not start on a new line. These elements flow within the content, allowing text and other inline elements to appear alongside them. Common inline elements are `span`, `anchor`, and `img` elements.
- **Block Level Elements**: Block-level elements start on a new line and take up the full width available to them by default, stretching across the width of their container. Some common block-level elements are `div`, `paragraph`, and `section` elements.
- **Inline-Block Level Elements**: You can set an element to `inline-block` by using the `display` property. These elements behave like inline elements but can have a `width` and `height` set like block-level elements.

Margin and Padding

- **`margin` Property**: This property is used to apply space outside the element, between the element's border and the surrounding elements.
- **`padding` Property**: This property is used to apply space inside the element, between the content and its border.
- **`margin` Shorthand**: You can specify 1–4 values to set the margin sides. One value applies to all four sides; two values set `top` and `bottom`, then `right` and `left`; three values set `top`, horizontal (`right` and `left`), then `bottom`; four values set `top`, `right`, `bottom`, `left`.
- **`padding` Shorthand**: You can specify 1–4 values to set the padding sides. One value applies to all four sides; two values set `top` and `bottom`, then `right` and `left`; three values set `top`, horizontal (`right` and `left`), then `bottom`; four values set `top`, `right`, `bottom`, `left`.

CSS Specificity

- **Inline CSS Specificity**: Inline CSS has the highest specificity because it is applied directly to the element. It overrides any internal or external CSS. The specificity value for inline styles is (1, 0, 0, 0).
- **Internal CSS Specificity**: Internal CSS is defined within a `style` element in the `head` section of the HTML document. It has lower specificity than inline styles but can override external styles.
- **External CSS Specificity**: External CSS is linked via a `link` element in the `head` section and is written in separate `.css` files. It has the lowest specificity but provides the best maintainability for larger projects.
- **Universal Selector (`*`)**: a special type of CSS selector that matches any element in the document. It is often used to apply a style to all elements on the page, which can be useful for resetting or normalizing styles across different browsers. The universal selector has the lowest specificity value of any selector. It contributes 0 to all parts of the specificity value (0, 0, 0, 0).
- **Type Selectors**: These selectors target elements based on their tag name. Type selectors have a relatively low specificity compared to other selectors. The specificity value for a type selector is (0, 0, 0, 1).
- **Class Selectors**: These selectors are defined by a period (`.`) followed by the class name. The specificity value for a class selector is (0, 0, 1, 0). This means that class selectors can override type selectors, but they can be overridden by ID selectors and inline styles.

- **ID Selectors**: ID selectors are defined by a hash symbol (`#`) followed by the ID name. ID selectors have a very high specificity, higher than type selectors and class selectors, but lower than inline styles. The specificity value for an ID selector is (0, 1, 0, 0).
- **`!important` keyword**: used to give a style rule the highest priority, allowing it to override any other declarations for a property. When used, it forces the browser to apply the specified style, regardless of the specificity of other selectors. You should be cautious when using `!important` because it can make your CSS harder to maintain and debug.
- **Cascade Algorithm**: An algorithm used to decide which CSS rules to apply when there are multiple styles targeting the same element. It ensures that the most appropriate styles are used, based on a set of well-defined rules.
- **CSS Inheritance**: The process by which styles are passed down from parent elements to their children. Inheritance allows you to define styles at a higher level in the document tree and have them apply to multiple elements without explicitly specifying them for each element.

--assignment--

Review the Basic CSS topics and concepts.

Basic HTML Review

HTML Basics

- **Role of HTML**: HTML represents the content and structure of the web page.
- **HTML Elements**: Elements are the building blocks for an HTML document. They represent headings, paragraphs, links, images and more. Most HTML elements consist of an opening tag (`<elementName>`) and a closing tag (`</elementName>`).

Here is the basic syntax:

```
<elementName>Content goes here</elementName>
```

- **Void Elements**: Void elements cannot have any content and only have a start tag. Examples include `img` and `meta` elements.

```
<img>
<meta>
```

It is common to see some codebases that include a forward slash `/` inside the void element. Both of these are acceptable:

```
<img>
<img/>
```

- **Attributes**: An attribute is a value placed inside the opening tag of an HTML element. Attributes provide additional information about the element or specify how the element should behave. Here is the basic syntax for an attribute:

```
<element attribute="value"></element>
```

A boolean attribute is an attribute that can either be present or absent in an HTML tag. If present, the value is true otherwise it is false. Examples of boolean attributes include `disabled`, `readonly`, and `required`.

- **Comments**: Comments are used in programming to leave notes for yourself and other developers in your code. Here is the syntax for a comment in HTML:

```
<!--This is an HTML comment.-->
```

Common HTML elements

- **Heading Elements**: There are six heading elements in HTML. The `h1` through `h6` heading elements are used to signify the importance of content below them. The lower the number, the higher the importance, so `h2` elements have less importance than `h1` elements.

```
<h1>most important heading element</h1>
<h2>second most important heading element</h2>
<h3>third most important heading element</h3>
<h4>fourth most important heading element</h4>
<h5>fifth most important heading element</h5>
<h6>least important heading element</h6>
```

- **Paragraph Elements**: This is used for paragraphs on a web page.

```
<p>This is a paragraph element.</p>
```

- **`img` Elements**: The `img` element is used to add images to the web page. The `src` attribute is used to specify the location for that image. For image elements, it's good practice to include another attribute called the `alt` attribute. Here's an example of an `img` element with the `src` and `alt` attributes:

```
<img src="https://cdn.freecodecamp.org/curriculum/cat-photo-app/lasagna.jpg"
alt="A slice of lasagna on a plate.">
```

- **`body` Element**: This element is used to represent the content for the HTML document.

```
<body>
  <h1>CatPhotoApp</h1>
  <p>This is a paragraph element.</p>
</body>
```

- **`section` Elements**: This element is used to divide up content into smaller sections.

```
<section>
  <h2>About Me</h2>
  <p>Hi, I am Jane Doe and I am a web developer.</p>
</section>
```

- **`div` Elements**: This element is a generic HTML element that does not hold any semantic meaning. It is used as a generic container to hold other HTML elements.

```
<div>
  <h1>I am a heading</h1>
  <p>I am a paragraph</p>
</div>
```

- **Anchor (`a`) Elements**: These elements are used to apply links to a web page. The `href` attribute is used to specify where the link should go when the user clicks on it.

```
<a href="https://cdn.freecodecamp.org/curriculum/cat-photo-app/running-
cats.jpg">cute cats</a>
```

- **Unordered (`ul`) and Ordered (`ol`) List Elements**: To create a bulleted list of items you should use the `ul` element with one or more `li` elements nested inside like this:

```
<ul>
  <li>catnip</li>
  <li>laser pointers</li>
  <li>lasagna</li>
</ul>
```

To create an ordered list of items you should use the `ol` element:

```
<ol>
  <li>flea treatment</li>
  <li>thunder</li>
  <li>other cats</li>
</ol>
```

- **Emphasis (`em`) Element**: This is used to place emphasis on a piece of text.

```
<p>Cats <em>love</em> lasagna.</p>
```

- **Strong Importance (`strong`) Element**: This element is used to place strong emphasis on text to indicate a sense of urgency and seriousness.

```
<p>
  <strong>Important:</strong> Before proceeding, make sure to wear your
safety goggles.
</p>
```

- **`figure` and `figcaption` Elements**: The `figure` element is used to group content like images and diagrams. The `figcaption` element is used to represent a caption for that content inside the `figure` element.

```
<figure>
  <img src="https://cdn.freecodecamp.org/curriculum/cat-photo-app/cats.jpg"
alt="Two tabby kittens sleeping together on a couch.">
  <figcaption>Cats <strong>hate</strong> other cats.</figcaption>
</figure>
```

- `main` **Element**: This element is used to represent the main content for a web page.
- `footer` **Element**: This element is placed at the bottom of the HTML document and usually contains copyright information and other important page links.

```
<footer>
  <p>
    No Copyright - <a
href="https://www.freecodecamp.org">freeCodeCamp.org</a>
  </p>
</footer>
```

Identifiers and Grouping

- **IDs**: Unique element identifiers for HTML elements. ID names should only be used once per HTML document.

```
<h1 id="title">Movie Review Page</h1>
```

ID names cannot have spaces. If your ID name contains multiple words then you can use dashes between the words like this:

```
<div id="red-box"></div>
```

- **Classes**: Classes are used to group elements for styling and behavior.

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

Unlike IDs, you can reuse the same class name throughout the HTML document. The `class` value can also have spaces like this:

```
<div class="box red-box"></div>
<div class="box blue-box"></div>
```

Special Characters and Linking

- **HTML entities**: An HTML entity, or character reference, is a set of characters used to represent a reserved character in HTML. Examples include the ampersand (`&`) symbol and the less than symbol (`<`).

```
<p>This is an &lt;img /&gt; element</p>
```

- `link` **Element**: This element is used to link to external resources like stylesheets and site icons. Here is the basic syntax for using the `link` element for an external CSS file:

```
<link rel="stylesheet" href="./styles.css" />
```

The `rel` attribute is used to specify the relationship between the linked resource and the HTML document. The `href` attribute is used to specify the location of the URL for the external resource.

- `script` **Element**: This element is used to embed executable code.

```
<body>
  <script>
    alert("Welcome to freeCodeCamp");
  </script>
</body>
```

While you can technically write all of your JavaScript code inside the `script` tags, it is considered best practice to link to an external JavaScript file instead. Here is an example of using the `script` element to link to an external JavaScript file:

```
<script src="path-to-javascript-file.js"></script>
```

The `src` attribute is used here to specify the location for that external JavaScript file.

Boilerplate and Encoding

- **HTML boilerplate**: This is a boilerplate that includes the basic structure and essential elements every HTML document needs.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>freeCodeCamp</title>
    <link rel="stylesheet" href="./styles.css" />
  </head>
  <body>
    <!--Headings, paragraphs, images, etc. go inside here-->
  </body>
</html>
```

- `DOCTYPE`: This is used to tell browsers which version of HTML you're using.
- `html` **Element**: This represents the top level element or the root of an HTML document. To specify the language for the document, you should use the `lang` attribute.
- `head` **Element**: The `head` section contains important meta data which is behind-the-scenes information needed for browsers and search engines.
- `meta` **Elements**: These elements represent your site's metadata. These element have details about things like character encoding, and how websites like Twitter should preview your page's link and more.
- `title` **Element**: This element is used to set the text that appears in the browser tab or window.
- **UTF-8 character encoding**: UTF-8, or UCS Transformation Format 8, is a standardized character encoding widely used on the web. Character encoding is the method computers use to store characters as data. The `charset` attribute is used inside of a `meta` element to set the character encoding to UTF-8.

SEO and Social Sharing

- **SEO**: Search Engine Optimization is a practice that optimizes web pages so they become more visible and rank higher on search engines.
- **Meta (`description`) Element**: This is used to provide a short description for the web page and impacting SEO.

```

```

```
<meta
  name="description"
  content="Discover expert tips and techniques for gardening in small spaces,
choosing the right plants, and maintaining a thriving garden."
/>
```

- **Open Graph tags**: The open graph protocol enables you to control how your website's content appears across various social media platforms, such as Facebook, LinkedIn, and many more. By setting these open graph properties, you can entice users to want to click and engage with your content. You can set these properties through a collection of `meta` elements inside your HTML `head` section.
- `og:title` **Property**: This is used to set the title that displays for social media posts.

```
<meta content="freeCodeCamp.org" property="og:title" />
```

- `og:type` **Property**: The `type` property is used to represent the type of content being shared on social media. Examples of this content include articles, websites, videos, or music.

```
<meta property="og:type" content="website" />
```

- `og:image` **Property**: This is used to set the image shown for social media posts.

```
<meta

content="https://cdn.freecodecamp.org/platform/universal/fcc_meta_1920X1080-
indigo.png"
  property="og:image"
/>
```

- `og:url` **Property**: This is used to set the URL that users will click on for the social media posts.

```
<meta property="og:url" content="https://www.freecodecamp.org" />
```

Media Elements and Optimization

- **Replaced elements**: A replaced element is an element whose content is determined by an external resource rather than by CSS itself. An example would be an `iframe` element. `iframe` stands for inline frame. It's an inline element used to embed other HTML content directly within the HTML page.

```
<iframe src="https://www.example.com" title="Example Site"></iframe>
```

You can include the `allowfullscreen` attribute which allows the user to display the iframe in full screen mode.

```
<iframe
  src="video-url"
  width="width-value"
  height="height-value"
```

```
  allowfullscreen
></iframe>
```

To embed a video within an `iframe` you can copy it directly from popular video services like YouTube and Vimeo, or define it yourself with the `src` attribute pointing to the URL of that video. Here's an example of embedding a popular freeCodeCamp course from YouTube:

```
<h1>A freeCodeCamp YouTube Video Embedded with the iframe Element</h1>

<iframe
  width="560"
  height="315"
  src="https://www.youtube.com/embed/PkZNo7MFNFg?si=-UBVIUNM3csdeiWF"
  title="YouTube video player"
  allow="accelerometer; autoplay; clipboard-write; encrypted-media;
gyroscope; picture-in-picture; web-share"
  referrerpolicy="strict-origin-when-cross-origin"
  allowfullscreen
></iframe>
```

There are some other replaced elements, such as `video`, and `embed`. And some elements behave as replaced elements under specific circumstances. Here's an example of an `input` element with the `type` attribute set to `image`:

```
<input type="image" alt="Descriptive text goes here" src="example-img-url">
```

- **Optimizing media**: There are three tools to consider when using media, such as images, on your web pages: the size, the format, and the compression. A compression algorithm is used to reduce the size for files or data.
- **Image formats**: Two of the most common file formats are PNG and JPG, but these are no longer the most ideal formats for serving images. Unless you need support for older browsers, you should consider using a more optimized format, like WEBP or AVIF.
- **Image licenses**: An image under the public domain has no copyright attached to it and is free to be used without any restrictions. Images licensed specifically under the Creative Commons 0 license are considered public domain. Some images might be released under a permissive license, like a Creative Commons license, or the BSD license that freeCodeCamp uses.
- **SVGs**: Scalable Vector Graphics track data based on paths and equations to plot points, lines, and curves. What this really means is that a vector graphic, like an SVG, can be scaled to any size without impacting the quality.

Multimedia Integration

- **`audio` and `video` Elements**: The `audio` and `video` elements allow you to add sound and video content to your HTML documents. The `audio` element supports popular audio formats like mp3, wav, and ogg. The `video` element supports mp4, ogg, and webm formats.

```
<audio src="CrystalizeThatInnerChild.mp3"></audio>
```

If you want to see the audio player on the page, then you can add the `audio` element with the `controls` attribute:

```
<audio src="CrystalizeThatInnerChild.mp3" controls></audio>
```

The `controls` attribute enables users to manage audio playback, including adjusting volume, and pausing, or resuming playback. The `controls` attribute is a boolean attribute that can be added to an element to enable built-in playback controls. If omitted, no controls will be shown.

- **`loop` Attribute**: The `loop` attribute is a boolean attribute that makes the audio replay continuously.

```
<audio
  src="https://cdn.freecodecamp.org/curriculum/js-music-player/can't-stay-down.mp3"
  loop
  controls
></audio>
```

- **`muted` Attribute**: When present in the `audio` element, the `muted` boolean attribute will start the audio in a muted state.

```
<audio
  src="https://cdn.freecodecamp.org/curriculum/js-music-player/can't-stay-down.mp3"
  loop
  controls
  muted
></audio>
```

- **`source` Element**: When it comes to audio file types, there are differences in which browsers support which type. To accommodate this, you can use `source` elements inside the `audio` element and the browser will select the first source that it understands. Here's an example of using multiple `source` elements for an `audio` element:

```
<audio controls>
  <source src="audio.ogg" type="audio/ogg" />
  <source src="audio.wav" type="audio/wav" />
  <source src="audio.mp3" type="audio/mpeg" />
</audio>
```

All the attributes we have learned so far are also supported in the `video` element. Here's an example of using a `video` element with the `loop`, `controls`, and `muted` attributes:

```
<video

src="https://archive.org/download/BigBuckBunny_124/Content/big_buck_bunny_720p_surround.mp4"
  loop
  controls
  muted
></video>
```

- **`poster` Attribute**: If you wanted to display an image while the video is downloading, you can use the `poster` attribute. This attribute is not available for `audio` elements and is unique to the `video` element.

```
<video

src="https://archive.org/download/BigBuckBunny_124/Content/big_buck_bunny_720
```

```
p_surround.mp4"
  loop
  controls
  muted
  poster="https://peach.blender.org/wp-content/uploads/title_anouncement.jpg?
x11217"
  width="620"
></video>
```

Target attribute types

- **`target` Attribute**: This attribute tells the browser where to open the URL for the anchor element. There are four important possible values for this attribute: `_self`, `_blank`, `_parent` and `_top`. There is a fifth value, called `_unfencedTop`, which is currently used for the experimental `FencedFrame` API. You probably won't have a reason to use this one yet.
- **`_self` Value**: This is the default value for the `target` attribute. This opens the link in the current browsing context. In most cases, this will be the current tab or window.

```
<a href="https://freecodecamp.org" target="_self">Visit freeCodeCamp</a>
```

- **`_blank` Value**: This opens the link in a new browsing context. Typically, this will open in a new tab. But some users might configure their browsers to open a new window instead.

```
<a href="https://freecodecamp.org" target="_blank">Visit freeCodeCamp</a>
```

- **`_parent` Value**: This opens the link in the parent of the current context. For example, if your website has an iframe, a `_parent` value in that iframe would open in your website's tab/window, not in the embedded frame.

```
<a href="https://freecodecamp.org" target="_parent">Visit freeCodeCamp</a>
```

- **`_top` Value**: This opens the link in the top-most browsing context - think "the parent of the parent". This is similar to `_parent`, but the link will always open in the full browser tab/window, even for nested embedded frames.

```
<a href="https://freecodecamp.org" target="_top">Visit freeCodeCamp</a>
```

Absolute vs. Relative Paths

- **Path definition**: A path is a string that specifies the location of a file or directory in a file system. In web development, paths let developers link to resources like images, stylesheets, scripts, and other web pages.
- **Path Syntax**: There are three key syntaxes to know. First is the slash, which can be a backslash (`\`) or forward slash (`/`) depending on your operating system. The second is the single dot (`.`). And finally, we have the double dot (`..`). The slash is known as the "path separator". It is used to indicate a break in the text between folder or file names. A single dot points to the current directory, and two dots point to the parent directory.

```
public/index.html
./favicon.ico
../src/index.css
```

- **Absolute Path**: An absolute path is a complete link to a resource. It starts from the root directory, includes every other directory, and finally the filename and extension. The "root directory" refers to the top-level directory or folder in a hierarchy. An absolute path also includes the protocol - which could be `http`, `https`, and `file` and the domain name if the resource is on the web. Here's an example of an absolute path that links to the freeCodeCamp logo:

```
<a href="https://design-style-
guide.freecodecamp.org/img/fcc_secondary_small.svg">
  View fCC Logo
</a>
```

- **Relative Path**: A relative path specifies the location of a file relative to the directory of the current file. It does not include the protocol or the domain name, making it shorter and more flexible for internal links within the same website. Here's an example of linking to the `about.html` page from the `contact.html` page, both of which are in the same folder:

```
<p>
  Read more on the
  <a href="about.html">About Page</a>
</p>
```

Link states

- `:link`: This is the default state. This state represents a link which the user has not visited, clicked, or interacted with yet. You can think of this state as providing the base styles for all links on your page. The other states build on top of it.
- `:visited`: This applies when a user has already visited the page being linked to. By default, this turns the link purple - but you can leverage CSS to provide a different visual indication to the user.
- `:hover`: This state applies when a user is hovering their cursor over a link. This state is helpful for providing extra attention to a link, to ensure a user actually intends to click it.
- `:focus`: This state applies when we focus on a link.
- `:active`: This state applies to links that are being activated by the user. This typically means clicking on the link with the primary mouse button by left clicking, in most cases.

--assignment--

Review the Basic HTML topics and concepts.

Classes and Objects Review

Python Classes and Objects

- **Class Definition**: A class is a blueprint for creating objects. It defines the behavior an object will have through its attributes and methods. Here is a basic example of a class definition in Python:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f'{self.name.upper()} says woof woof!')
```

- **Creating Objects**: Objects are instances of a class. They are created by calling the class with the necessary arguments.

```
dog1 = Dog('Jack', 3)
dog2 = Dog('Thatcher', 5)

dog1.bark()  # JACK says woof woof!
dog2.bark()  # THATCHER says woof woof!
```

- **Calling methods with objects**: You can call methods on objects to perform actions or retrieve information.

```
objectName1.methodName()
objectName2.methodName()
```

- **Difference Between Class and Object**: A class is a reusable template, while an object is a specific instance of that class with actual data.

Attributes

- **Instance Attributes**: Defined in `__init__()` using `self`, and unique to each object.
- **Class Attributes**: Defined directly inside the class and shared by all instances.

```
class Dog:
    species = 'French Bulldog'  # Class attribute

    def __init__(self, name):
        self.name = name  # Instance attribute

print(Dog.species) # French Bulldog

jack = Dog('Jack')
print(jack.name)     # Jack
print(jack.species)  # French Bulldog
```

Methods

- **Methods**: Functions defined inside a class that operate on the object's attributes.

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model

    def describe(self):
        return f'This car is a {self.color} {self.model}'

my_car_1 = Car('red', 'Tesla Model S')
print(my_car_1.describe())  # This car is a red Tesla Model S
```

- **Accessing Methods**: Call methods on objects using the dot notation. Here is an example of calling the `describe` method on two different car objects:

```
class Car:
 def __init__(self, color, model):
   self.color = color
   self.model = model

 def describe(self):
   return f'This car is a {self.color} {self.model}'

my_car_1 = Car('red', 'Tesla Model S')
my_car_2 = Car('green', 'Lamborghini Revuelto')

print(my_car_1.describe()) # Calling methods using the dot notation

print(my_car_2.describe()) # Calling methods using the dot notation
```

Dunder (Magic) Methods

- **Definition**: Special methods that start and end with a double underscore (e.g., `__init__`, `__len__`, `__str__`, `__eq__`). Python uses them internally for built-in operations.

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __len__(self):
        return self.pages

    def __str__(self):
        return f"'{self.title}' has {self.pages} pages"

    def __eq__(self, other):
        return self.pages == other.pages

book1 = Book('Built Wealth Like a Boss', 420)
print(len(book1))        # 420
print(str(book1))        # 'Built Wealth Like a Boss' has 420 pages
```

- **Calling dunder methods indirectly**: You don't need to call dunder methods directly. Instead, Python automatically calls them when certain actions happen. These operations include:
    - **arithmetic operations like addition, subtraction, multiplication, division, and others**. In addition, `__add__()` is called, `__sub__()` for subtraction, `__mul__()` for multiplication, and `__truediv__()` for division.
    - **string operations like concatenation, repetition, formatting, and conversion to text**. `__add__()` is called for concatenation, `__mul__()` for repetition, `__format__()` for formatting, `__str__()` and `__repr__()` for text conversion, and so on.
    - **comparison operations like equality, less-than, greater-than, and others**. `__eq__()` is called for equality checks, `__lt__()` for less-than, `__gt__()` for greater-than, and so on.
    - **iteration operations like making an object iterable and advancing through items**. `__iter__()` is called to return an iterator and `__next__()` to fetch the next item.

Real World Example: Shopping Cart

- **Cart Class with Dunder Methods**: Allows adding, removing, iterating, and checking contents with built-in behavior.

```
class Cart:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

    def remove(self, item):
        if item in self.items:
            self.items.remove(item)
        else:
            print(f'{item} is not in cart')

    def list_items(self):
        return self.items

    def __len__(self):
        return len(self.items)

    def __getitem__(self, index):
        return self.items[index]

    def __contains__(self, item):
        return item in self.items

    def __iter__(self):
        return iter(self.items)

cart = Cart()
cart.add('Laptop')
print(len(cart))        # 1
print('Laptop' in cart) # True
```

--assignment--

Review the Classes and Objects topics and concepts.

Computer Basics Review

Understanding Computer, Internet and Developer Tooling Basics

- **Motherboard**: holds all of the memory, connectors, and hard drives that are needed to run the computer. It serves as the main circuit board for the computer.
- **Central Processing Unit(CPU)**: a processor that is responsible for executing instructions and performing calculations.
- **Random Access Memory(RAM)**: a temporary storage location for the computer's CPU.
- **Hard Disk Drive(HDD)**: a permanent storage location that is used to store data even when the computer is turned off.
- **Solid State Drive(SSD)**: non-volatile flash memory and can be used in place of a hard drive.
- **Power Supply Unit(PSU)**: responsible for converting the electricity from the wall outlet into a form that the computer can use.
- **Graphics Processing Unit(GPU)**: responsible for rendering visuals on the computer screen.
- **Different Types of Internet Service Providers**: An Internet Service Provider (ISP) is a company that provides access to the internet. There are different types of ISPs, including dial-up, DSL, cable, fiber-optic, and satellite.
- **Safe Ways to Sign Into Your Computer**: Examples of safe ways to sign into your computer include using a strong password, enabling two-factor authentication, and using a password manager.

- **Integrated Development Environment (IDE)**: a tool that helps developers write, test and debug code in an efficient manner.
- **Code Editor**: a tool that developers use to write and debug code.
- **Git**: a popular version control system that allows developers to track changes in their code and collaborate with others.
- **Cloud-based Hosting Services for repositories**: A repository is a storage location for project files and version history. Popular cloud-based hosting services for repositories include GitHub, GitLab, and Bitbucket.
- **Package Managers**: tools that help developers simplify the process of adding, updating, and removing libraries and project dependencies. Examples include npm, pip, and Maven.
- **Testing Libraries and Frameworks**: Testing is done in software to ensure that the code works as expected. Examples of testing libraries and frameworks include Jest, PHPUnit, and JUnit.

Working With Files, File Systems and Media Formats

- **Best practices for naming files**: You will want to name your files in a way that is easy to understand and maintain. For example, `about-us.html` is a more descriptive name than `page1.html`.
- **root directory**: top-level directory in a file system. Directory is another name for a folder.
- **Markdown**: a markup language commonly used for documentation and `README` files. A `README` file is a file that contains information about a project, such as how to install and use it.
- **`index.html`**: represents the default page that is displayed when a user visits a website.
- **Create, Move, and Delete files and folders using Explorer/Finder**: Explorer is the file manager in Windows, and Finder is the file manager in macOS. You can use these tools to create, move, and delete files and folders.
- **Searching for files and folders**: You can use the search functionality in Explorer or Finder to find files and folders on your computer.
- **HTML, CSS and JS File Types**: `.html` file extension is used for HTML files, `.css` for CSS files, and `.js` for JavaScript files.
- **Common Image and Graphic Formats**: `JPEG` and `PNG` are common image file formats. `GIF` is another common image file format that supports animation. `SVG` is a file format for vector graphics.
- **Common Audio and Video Formats**: The `MP3` format is commonly used for audio files. The `MP4` format is commonly used for video files. The `MOV` format was developed by Apple and is commonly used for video files.
- **Common Font Formats**: The `TTF` format is commonly used for TrueType fonts. The `WOFF` format is commonly used for web fonts. The successor to WOFF is `WOFF2`, which provides better compression.
- **ZIP**: a file format that is used to compress files and folders.

Browsing the Web Effectively

- **What is a Web browser?**: a software application that allows users to access and view websites on the internet.
- **What is a Search Engine**: a tool that allows users to search for information on the internet. Examples include Google, Bing, and Yahoo.
- **Common web browsers**: A few examples of common web browsers include Google Chrome, Mozilla Firefox, and Microsoft Edge.
- **Common Search Strategies**: You can use `site:` followed by the URL of a website to search for content on that website. You can use `filetype:` followed by a file extension to search for files of that type. You can prefix a search term with a minus sign to exclude results containing that term. You can prefix a search term with a plus sign to include results containing that term.

--assignment--

Review the Computer Basics topics and concepts.

CSS Review

Review the concepts below to prepare for the upcoming exam.

CSS Basics

- **What is CSS?**: Cascading Style Sheets (CSS) is a markup language used to apply styles to HTML elements. CSS is used for colors, background images, layouts and more.
- **Basic Anatomy of a CSS Rule**: A CSS rule is made up of two main parts: a selector and a declaration block. A selector is a pattern used in CSS to identify and target specific HTML elements for styling. A declaration block applies a set of styles for a given selector, or selectors.
- **`meta name="viewport"` Element**: This `meta` element gives the browser instructions on how to control the page's dimensions and scaling on different devices, particularly on mobile phones and tablets.
- **Default Browser Styles**: Each HTML element will have default browser styles applied to them. This usually includes items like default margins and paddings.

Inline, Internal, and External CSS

- **Inline CSS**: These styles are written directly within an HTML element using the `style` attribute. Most of the time you will not be using inline CSS due to separation of concerns.
- **Internal CSS**: These styles are written within the `<style>` tags inside the `head` section of an HTML document. This can be useful for creating short code examples, but usually, you will not use internal CSS.
- **External CSS**: These styles are written in a separate CSS file and linked to the HTML document using the `link` element in the `head` section. For most projects, you will use an external CSS file over internal or inline CSS.

Working With the `width` and `height` Properties

- **`width` Property**: This property specifies the width of an element. If you do not specify a width, then the default is set to `auto`. This means the element will take up the full width of its parent container.
- **`min-width` Property**: This property specifies the minimum width for an element.
- **`max-width` Property**: This property specifies the maximum width for an element.
- **`height` Property**: This property specifies the height of an element. Similarly, the height is `auto` by default, which means it will adjust to the content inside.
- **`min-height` Property**: This property specifies the minimum height for an element.
- **`max-height` Property**: This property specifies the maximum height for an element.

Different Types of CSS Combinators

- **Descendant Combinator**: This combinator is used to target elements that are descendants of a specified parent element.
- **Child Combinator (`>`)**: This combinator is used to select elements that are direct children of a specified parent element.
- **Next-sibling Combinator (`+`)**: This combinator selects an element that immediately follows a specified sibling element.
- **Subsequent-sibling Combinator (`~`)**: This combinator selects all siblings of a specified element that come after it.

Inline, Block, and Inline-Block Level Elements

- **Inline Level Elements**: Inline elements only take up as much width as they need and do not start on a new line. These elements flow within the content, allowing text and other inline elements to appear alongside them. Common inline elements are `span`, `anchor`, and `img` elements.
- **Block Level Elements**: Block level elements that take up the full width available to them by default, stretching across the width of their container. Some common block-level elements are `div`, `paragraph`, and `section` elements.

- **Inline-Block Level Elements**: You can set an element to `inline-block` by using the `display` property. These elements behave like inline elements but can have a `width` and `height` set like block-level elements.

Margin and Padding

- **`margin` Property**: This property is used to apply space outside the element, between the element's border and the surrounding elements.
- **`margin` Shorthand**: You can specify 1–4 values to set the margin sides. One value applies to all four sides; two values set `top` and `bottom`, then `right` and `left`; three values set `top`, horizontal (`right` and `left`), then `bottom`; four values set `top`, `right`, `bottom`, `left`.
- **`padding` Property**: This property is used to apply space inside the element, between the content and its border.
- **`padding` Shorthand**: You can specify 1–4 values to set the padding sides. One value applies to all four sides; two values set `top` and `bottom`, then `right` and `left`; three values set `top`, horizontal (`right` and `left`), then `bottom`; four values set `top`, `right`, `bottom`, `left`.

CSS Specificity

- **Inline CSS Specificity**: Inline CSS has the highest specificity because it is applied directly to the element. It overrides any internal or external CSS. The specificity value for inline styles is (1, 0, 0, 0).
- **Internal CSS Specificity**: Internal CSS is defined within a `style` element in the `head` section of the HTML document. It has lower specificity than inline styles but can override external styles.
- **External CSS Specificity**: External CSS is linked via a `link` element in the `head` section and is written in separate `.css` files. It has the lowest specificity but provides the best maintainability for larger projects.
- **Universal Selector (`*`)**: a special type of CSS selector that matches any element in the document. It is often used to apply a style to all elements on the page, which can be useful for resetting or normalizing styles across different browsers. The universal selector has the lowest specificity value of any selector. It contributes 0 to all parts of the specificity value (0, 0, 0, 0).
- **Type Selectors**: These selectors target elements based on their tag name. Type selectors have a relatively low specificity compared to other selectors. The specificity value for a type selector is (0, 0, 0, 1).
- **Class Selectors**: These selectors are defined by a period (`.`) followed by the class name. The specificity value for a class selector is (0, 0, 1, 0). This means that class selectors can override type selectors, but they can be overridden by ID selectors and inline styles.
- **ID Selectors**: ID selectors are defined by a hash symbol (`#`) followed by the ID name. ID selectors have a very high specificity, higher than type selectors and class selectors, but lower than inline styles. The specificity value for an ID selector is (0, 1, 0, 0).
- **`!important` keyword**: used to give a style rule the highest priority, allowing it to override any other declarations for a property. When used, it forces the browser to apply the specified style, regardless of the specificity of other selectors. You should be cautious when using `!important` because it can make your CSS harder to maintain and debug.
- **Cascade Algorithm**: An algorithm used to decide which CSS rules to apply when there are multiple styles targeting the same element. It ensures that the most appropriate styles are used, based on a set of well-defined rules.
- **CSS Inheritance**: The process by which styles are passed down from parent elements to their children. Inheritance allows you to define styles at a higher level in the document tree and have them apply to multiple elements without explicitly specifying them for each element.

Design Terminology

- **Layout**: This is how visual elements are arranged on a page or screen to communicate a message. These elements may include text, images, and white space.
- **Alignment**: This is how the elements are placed in relation to one another. Using alignment correctly is helpful for making the design look clean and organized.
- **Composition**: This is the act of arranging elements to create a harmonious design. It determines how elements like images, text, and shapes relate to each other and contribute to the design in an

artistic way.
- **Balance**: This is how visual weight is distributed within a composition. Designers aim to create an equilibrium through symmetrical or asymmetrical arrangements.
- **Scale**: This refers to comparing the dimensions or size of one element to that of another.
- **Hierarchy**: This establishes the order of importance of the elements in a design. It's about making sure that the most important information is noticed first.
- **Contrast**: This is the process of creating clear distinctions between the elements. You can do this through variations in color, size, shape, texture, or any other visual characteristic. Strong contrast is also helpful for improving readability.
- **White Space(negative space)**: This is the empty space in a design. It's the area surrounding the elements.
- **UI(User Interface)**: UI includes the visual and interactive elements that users can see on their screens, like icons, images, text, menus, links, and buttons.
- **UX(User Experience)**: UX is about how users feel when using a product or service. An application with a well-designed user experience is intuitive, easy to use, efficient, accessible, and enjoyable.
- **Design Brief**: This is a document that outlines the objectives, goals, and requirements of a project. It is a roadmap that guides the design process and ensures that the final product meets the needs of the client.
- **Vector Based Design**: This involves creating digital art using mathematical formulas to define lines, shapes, and colors.
- **Prototyping**: This refers to the process of creating an interactive model of a product or user interface.

UI Design Fundamentals

- **Good Contrast for Background and Foreground Colors**: It is important to ensure that the background and foreground colors have good contrast to make the text readable. The Web Content Accessibility Guidelines (WCAG) recommend a minimum contrast ratio of 4.5:1 for normal text and 3:1 for large text.
- **Good Visual Hierarchy in Design**: A strong visual hierarchy can provide a clear path for the eye to follow, ensuring that the information you convey is consumed in the order that you intend.
- **Responsive Images**: Responsive images are images that scale to fit the size of the screen they are being viewed on. This is important because it ensures that your images look good on all devices, from desktops to mobile phones.
- **Progressive Enhancement**: This is a design approach that ensures all users, regardless of browser or device, can access the essential content and functionality of an application.
- **User-centered Design**: This is an approach that prioritizes the end user, from their needs to their preferences and limitations. The goal of user-centered design is to craft a web page that is intuitive, efficient to use, and pleasing for your users to interact with.
- **User Research**: This is the systematic study of the people who use your product. The goal is to measure user needs, behaviors, and pain points.
- **User Testing**: This refers to the practice of capturing data from users as they interface with your application.
- **A/B Testing**: This is the process of shipping a new feature to a randomly selected subset of your user base. You can then leverage analytics data to determine if the feature is beneficial.
- **User Requirements**: This refers to the stories or rubric that your application needs to follow. User requirements might be defined by user research, or industry standards. They can even be defined by stakeholder input.
- **Progressive Disclosure**: This is a design pattern used to only show users relevant content based on their current activity and hide the rest. This is done to reduce cognitive load and make the user experience more intuitive.
- **Deferred/Lazy Registration**: This is a UI design pattern that allows users to browse and interact with your application without having to register.

Design Best Practices

- **Dark Mode**: This is a special feature on web applications where you can change the default light color scheme to a dark color scheme. You should use desaturated colors in dark mode. Desaturated colors are colors that are less intense and have a lower saturation level.
- **Breadcrumbs**: This is a navigation aid that shows the user where they are in the site's hierarchy. It is best to place breadcrumbs at the top of the page so users can easily find it. Also, you want to make sure the breadcrumbs are large enough to be easily read, but not so large that they take up too much space on the page.
- **Card Component**: Your card component should be simple in design, not visually cluttered or display too much information. For media, make sure to choose high-quality images and videos to enhance the user experience.
- **Infinite Scroll**: This is a design pattern that loads more content as the user scrolls down the page. You should consider using a load more button because it gives a user control over when they want to see more content. You can also add a back button so users have the ability to go back to the previous page without having to scroll all the way back up.
- **Modal Dialog**: This is a type of pop-up that will display on top of existing page content. Usually the background content will have a dim color overlay in order to help the user better focus on the modal content. Also, it is always a good idea to allow the user to click outside of the modal to close it. When you use the HTML `dialog` element, you will get a lot of the functionality and accessibility benefits built in.
- **Progress Indication for Form Registration**: This is a way to show users how far they are in a process. It can be used in forms, registration, and setup processes. Your design should be simple, easy to find, and make it possible to go back to previous steps.
- **Shopping Cart**: Carts are a place for user to see what item they have already selected on an e-commerce platform. Your carts should always be visible to the user, use a common icon like a cart, bag or basket, and have a clear call-to-action button for users to proceed to checkout.

Common Design Tools

- **Figma**: This cloud-based tool specializes in User Interface and User Experience (UI / UX) design. It enables design and development teams to collaborate from anywhere, offering built-in features including Vector-based design, automatic layout, a commenting and feedback system and more.
- **Sketch**: This is a popular design tool used for its intuitive interface and simplicity, making it ideal for developers who want to quickly create prototypes. It's also widely used by designers for tasks like creating UIs, icons, and web layouts.
- **Adobe XD**: This is a vector-based design and prototyping tool for UI/UX design, known for its seamless integration with other Adobe apps like Photoshop, Illustrator, and After Effects.
- **Canva**: This tool allows you to create a wide range of visual content, including posters, cover photos, presentations, short videos, and more. Its user-friendly and simple design makes it ideal for beginners.

Absolute Units

- `px` **(Pixels)**: This absolute unit is a fixed-size unit of measurement in CSS, providing precise control over dimensions. This means that 1px is always equal to 1/96th of an inch.
- `in` **(Inch)**: This absolute unit is equal to 96px.
- `cm` **(Centimeters)**: This absolute unit is equal to 25.2/64 of an inch.
- `mm` **(Millimeters)**: This absolute unit is equal to 1/10th of a centimeter.
- `q` **(Quarter-Millimeters)**: This absolute unit is equal to 1/40th of a centimeter.
- `pc` **(Picas)**: This absolute unit is equal to 1/6th of an inch.
- `pt` **(Points)**: This absolute unit is equal to 1/72th of an inch.

Relative Units

- **Percentages**: These relative units allow you to define sizes, dimensions, and other properties as a proportion of their parent element. For example, if you set `width: 50%;` on an element, it will occupy half the width of its parent container.
- `em` **Unit**: These units are relative to the font size of the element. If you are using `ems` for the `font-size` property, the size of the text will be relative to the font size of the parent element.

- **`rem` Unit**: These units are relative to the font size of the root element, which is the `html` element.
- **`vh` Unit**: `vh` stands for `"viewport height"` and `1vh` is equal to 1% of the viewport's height.
- **`vw` Unit**: `vw` stands for `"viewport width"` and `1vw` is equal to 1% of the viewport's width.

`calc` Function

- **`calc()` Function**: With the `calc()` function, you can perform calculations directly within your stylesheets to determine property values dynamically. This means that you can create flexible and responsive user interfaces by calculating dimensions based on the viewport size or other elements.

User Action Pseudo-classes

- **Pseudo-classes Definition**: These are special CSS keywords that allow you to select an element based on its specific state or position.
- **User Action Pseudo-classes**: These are special keywords that allow you to change the appearance of elements based on user interactions, improving the overall user experience.
- **`:active` Pseudo-class**: This pseudo-class lets you select the active state of an element, like clicking on a button.
- **`:hover` Pseudo-class**: This pseudo-class defines the hover state of an element.
- **`:focus` Pseudo-class**: This pseudo-class applies styles when an element gains focus, typically through keyboard navigation or when a user clicks into a form input.
- **`:focus-within` Pseudo-class**: This pseudo-class is used to apply styles to an element when it or any of its descendants have focus.

Input Pseudo-classes

- **Input Pseudo-classes**: These pseudo-classes are used to target HTML `input` elements based on the state they are in before and after user interaction.
- **`:enabled` Pseudo-class**: This pseudo-class is used to target form buttons or other elements that are currently enabled.
- **`:disabled` Pseudo-class**: This pseudo-class lets you style an interactive element in disabled mode.
- **`:checked` Pseudo-class**: This pseudo-class is used to indicate to the user that it is checked.
- **`:valid` Pseudo-class**: This pseudo-class targets the input fields that meet the validation criteria.
- **`:invalid` Pseudo-class**: This pseudo-class targets the input fields that do not meet the validation criteria.
- **`:in-range` and `:out-of-range` Pseudo-classes**: These pseudo-classes applies styles to elements based on whether their values are within or outside specified range constraints.
- **`:required` Pseudo-class**: This pseudo-class targets `input` elements that have the `required` attribute. It signals to the user that they must fill out the field to submit the form.
- **`:optional` Pseudo-class**: This pseudo-class applies styles input elements that are not required and can be left empty.
- **`:autofill` Pseudo-class**: This pseudo-class applies styles to input fields that the browser automatically fills with saved data.

Location Pseudo-classes

- **Location Pseudo-classes**: These pseudo-classes are used for styling links and elements that are targeted within the current document.
- **`:any-link` Pseudo-class**: This pseudo-class is a combination of the `:link` and `:visited` pseudo-classes. So, it matches any anchor element with an href attribute, regardless of whether it's visited or not.
- **`:link` Pseudo-class**: This pseudo-class allows you to target all unvisited links on a webpage. You can use it to style links differently before the user clicks on them.
- **`:local-link` Pseudo-class**: This pseudo-class targets links that point to the same document. It can be useful when you want to differentiate internal links from external ones.
- **`:visited` Pseudo-class**: This pseudo-class targets a link the user has visited.
- **`:target` Pseudo-class**: This pseudo-class is used to apply styles to an element that is the target of a URL fragment

Tree-structural Pseudo-classes

- **Tree-structural Pseudo-classes**: These pseudo-classes allow you to target and style elements based on their position within the document tree.
- `:root` **Pseudo-class**: This pseudo-class is usually the root `html` element. It helps you target the highest level in the document so you can apply a common style to the entire document.
- `:empty` **Pseudo-class**: Empty elements, that is, elements with no children other than white space, are also included in the document tree. That's why there's an `:empty` pseudo-class to target empty elements.
- `:nth-child(n)` **Pseudo-class**: This pseudo-class allows you to select elements based on their position within a parent.
- `:nth-last-child(n)` **Pseudo-class**: This pseudo-class enables you to select elements by counting from the end.
- `:first-child` **Pseudo-class**: This pseudo-class selects the first element in a parent element or the document.
- `:last-child` **Pseudo-class**: This pseudo-class selects the last element in a parent element or the document
- `:only-child` **Pseudo-class**: This pseudo-class selects the only element in a parent element or the document
- `:first-of-type` **Pseudo-class**: This pseudo-class selects the first occurrence of a specific element type within its parent.
- `:last-of-type` **Pseudo-class**: This pseudo-class selects the last occurrence of a specific element type within its parent.
- `:nth-of-type(n)` **Pseudo-class**: This pseudo-class allows you to select a specific element within its parent based on its position among siblings of the same type.
- `:only-of-type` **Pseudo-class**: This pseudo-class selects an element if it's the only one of its type within its parent.

Functional Pseudo-classes

- **Functional Pseudo-classes**: Functional pseudo-classes allow you to select elements based on more complex conditions or relationships. Unlike regular pseudo-classes which target elements based on a state (for example, `:hover`, `:focus`), functional pseudo-classes accept arguments.
- `:is()` **Pseudo-class**: This pseudo-class takes a list of selectors (ex. `ol`, `ul`) and selects an element that matches one of the selectors in the list.
- `:where()` **Pseudo-class**: This pseudo-class takes a list of selectors (ex. `ol`, `ul`) and selects an element that matches one of the selectors in the list. The difference between `:is` and `:where` is that the latter will have a specificity of 0.
- `:has()` **Pseudo-class**: This pseudo-class is often dubbed the `"parent"` selector because it allows you to style elements who contain child elements specified in the selector list.

Pseudo-elements

- `::before` **Pseudo-element**: This pseudo-element uses the `content` property to insert cosmetic content like icons just before the element.
- `::after` **Pseudo-element**: This pseudo-element uses the `content` property to insert cosmetic content like icons just after the element.
- `::first-letter` **Pseudo-element**: This pseudo-element targets the first letter of an element's content, allowing you to style it.
- `::marker` **Pseudo-element**: This pseudo-element lets you select the marker (bullet or numbering) of list items for styling.

Color Theory

- **Color Theory Definition**: This is the study of how colors interact with each other and how they affect our perception. It covers color relationships, color harmony, and the psychological impact of color.
- **Primary Colors**: These colors which are yellow, blue, and red, are the fundamental hues from which all other colors are derived.

- **Secondary Colors**: These colors result from mixing equal amounts of two primary colors. Green, orange, and purple are examples of secondary colors.
- **Tertiary Colors**: These colors result from combining a primary color with a neighboring secondary color. Yellow-Green, Blue-Green, and Blue-Violet, are examples of tertiary colors.
- **Warm Colors**: These colors which include reds, oranges, and yellows, evoke feelings of comfort, warmth, and coziness.
- **Cool Colors**: These colors which include blues, green, and purples, evoke feelings of calmness, serenity, and professionalism.
- **Color Wheel**: The color wheel is a circular diagram that shows how colors relate to each other. It's an essential tool for designers because it helps them to select color combinations.
- **Analogous Color Schemes**: These color schemes create cohesive and soothing experiences. They have analogous colors, which are adjacent to each other in the color wheel.
- **Complementary Color Schemes**: These color schemes create high contrast and visual impact. Their colors are located on the opposite ends of the color wheel, relative to each other.
- **Triadic Color Scheme**: This color scheme has vibrant colors. They are made from colors that are approximately equidistant from each other. If they are connected, they form an equilateral triangle on the color wheel.
- **Monochromatic Color Scheme**: For this color scheme, all the colors are derived from the same base color by adjusting its lightness, darkness, and saturation. This evokes a feeling of unity and harmony while still creating contrast.

Different Ways to Work with Colors in CSS

- **Named Colors**: These colors are predefined color names recognized by browsers. Examples include `blue`, `darkred`, `lightgreen`.
- **`rgb()` Function**: RGB stands for Red, Green, and Blue — the primary colors of light. These three colors are combined in different intensities to create a wide range of colors. the `rgb()` function allows you to define colors using the RGB color model.
- **`rgba()` Function**: This function adds a fourth value —alpha— that controls the transparency of the color.
- **`hsl()` Function**: HSL stands for Hue, Saturation, and Lightness — three key components that define a color.
- **`hsla()` Function**: This function adds a fourth value -alpha- that controls the opacity of the color.
- **Hexadecimal**: A hex code (short for hexadecimal code) is a six-character string used to represent colors in the RGB color model. The "hex" refers to the base-16 numbering system, which uses digits 0 to 9 and letters A to F.

Linear and Radial Gradients

- **Linear Gradients**: These gradients create a gradual blend between colors along a straight line. You can control the direction of this line and the colors used.
- **Radial Gradients**: These gradients create circular or elliptical gradients that radiate from a central point.

Best Practices for Styling Inputs

- **Styling Inputs**: As with all text elements, you need to ensure the styles you apply to text inputs are accessible. This means the font needs to be adequately sized, and the color needs to have sufficient contrast with the background. Input elements are also focusable. When you are editing your styles, you should take care that you preserve a noticeable indicator when the element has focus, such as a bold border.

Using `appearance: none` for Inputs

- **`appearance: none`**: Browsers apply default styling to a lot of elements. The `appearance: none` CSS property gives you complete control over the styling, but comes with some caveats. When building custom styles for input elements, you will need to make sure focus and error indicators are still present.

Common Issues Styling `datetime-local` and `color` Properties

- **Common Issues**: These special types of inputs rely on complex pseudo-elements to create things like the date and color pickers. This presents a significant challenge for styling these inputs. One challenge is that the default styling is entirely browser-dependent, so the CSS you write to make the picker look the way you intend may be entirely different on another browser.

CSS Overflow Property

- **Definition**: Overflow refers to the way elements handle content that exceeds, or "overflows", the size of the containing element. Overflow is two-dimensional.
- **`overflow-x`**: The x-axis determines horizontal overflow.
- **`overflow-y`**: the y-axis determines vertical overflow.

CSS Transform Property

- **Definition**: This property enables you to apply various transformations to elements, such as rotating, scaling, skewing, or translating (moving) them in 2D or 3D space.
- **`translate()` Function**: This function is used to move an element from its current position.
- **`scale()` function**: This function allows you to change the size of an element.
- **Transforms and Accessibility**: If you're using transform to hide or reveal content, make sure that the content is still accessible to screen readers and keyboard navigation. Hidden content should be truly hidden, such as by using `display: none` or `visibility: hidden`, rather than simply being visually moved off-screen.

The Box Model

- **Definition**: In the CSS box model, every element is surrounded by a box. This box consists of four components: the content area, `padding`, `border`, `margin`.
- **Content Area**: The content area is the innermost part of the box. It's the space that contains the actual content of an element, like text or images.
- **`padding`**: The padding is the area immediately after the content area. It's the space between the content area and the border of an element.
- **`border`**: The border is the outer edge or outline of an element in the CSS box model. It's the visual boundary of the element.
- **`margin`**: The margin is the space outside the border of an element. It determines the distance between an element and other elements around it.

Margin Collapse

- **Definition**: This behavior occurs when the vertical margins of adjacent elements overlap, resulting in a single margin equal to the larger of the two. This behavior applies only to vertical margins (top and bottom), not horizontal margins (left and right).

The `content-box` and `border-box` Property Values

- **`box-sizing` Property**: This property is used to determine how the final `width` and `height` are calculated for an HTML element.
- **`content-box` Value**: In the `content-box` model, the `width` and `height` that you set for an element determine the dimensions of the content area but they don't include the `padding`, `border`, or `margin`.
- **`border-box` Value**: With `border-box`, the `width` and `height` of an element include the content area, the `padding`, and the `border`, but they don't include the `margin`.

CSS Reset

- **Definition**: A CSS reset is a stylesheet that removes all or some of the default formatting that web browsers apply to HTML elements. Third party options for CSS resets include `sanitize.css` and `normalize.css`.

CSS Filter Property

- **Definition**: This property can be used to create various effects such as blurring, color shifting, and contrast adjustment.
- `blur()` **Function**: This function applies a Gaussian blur to the element. The amount is defined in pixels and represents the radius of the blur.
- `brightness()` **Function**: This function adjusts the brightness of the element. A value of 0% will make the element completely black, while values over 100% will increase the brightness.
- `grayscale()` **Function**: This function converts the element to grayscale. The amount is defined as a percentage, where 100% is completely grayscale and 0% leaves the image unchanged.
- `sepia()` **Function**: This function applies a sepia tone to the element. Like grayscale, it uses a percentage value.
- `hue-rotate()` **Function**: This function applies a hue rotation to the element. The value is defined in degrees and represents a rotation around the color circle.

Introduction to CSS Flexbox and Flex Model

- **Definition**: CSS flexbox is a one-dimensional layout model that allows you to arrange elements in rows and columns within a container.
- **Flex Model**: This model defines how flex items are arranged within a flex container. Every flex container has two axes: the main axis and the cross axis.

The `flex-direction` Property

- **Definition**: This property sets the direction of the main axis. The default value of `flex-direction` is `row`, which places all the flex items on the same row, in the direction of your browser's default language (left to right or right to left).
- `flex-direction: row-reverse;`: This reverses the items in the row.
- `flex-direction: column;`: This will align the flex items vertically instead of horizontally.
- `flex-direction: column-reverse;`: This reverses the order of the flex items vertically.

The `flex-wrap` Property

- **Definition**: This property determines how flex items are wrapped within a flex container to fit the available space. `flex-wrap` can take three possible values: `nowrap`, `wrap`, and `wrap-reverse`.
- `flex-wrap: nowrap;`: This is the default value. Flex items won't be wrapped onto a new line, even if their width exceed the container's width.
- `flex-wrap: wrap;`: This property will wrap the items when they exceed the width of their container.
- `flex-wrap: wrap-reverse;`: This property will wrap flex items in reverse order.
- `flex-flow` **Property**: This property is a shorthand property for `flex-direction` and `flex-wrap`.

The `justify-content` Property

- **Definition**: This property aligns the child elements along the main axis of the flex container.
- `justify-content: flex-start;`: In this case, the flex items will be aligned to the start of the main axis. This could be horizontal or vertical.
- `justify-content: flex-end;`: In this case, the flex items are aligned to the end of the main axis, horizontally or vertically.
- `justify-content: center;`: This centers the flex items along the main axis.
- `justify-content: space-between;`: This will distribute the elements evenly along the main axis.
- `justify-content: space-around;`: This will distribute flex items evenly within the main axis, adding a space before the first item and after the last item.
- `justify-content: space-evenly;`: This will distributes the items evenly along the main axis.

The `align-items` Property

- **Definition**: This property is used to distribute items along the cross axis. Remember that the cross axis is perpendicular to the main axis.

- `align-items: center;`: This is used to center the items along the cross axis.
- `align-items: flex-start;`: This aligns the items to the start of the cross axis.
- `align-items: stretch;`: This is used to stretch the flex items along the cross axis.

Introduction to Typography

- **Definition**: Typography is the art of choosing the right fonts and format to make text visually appealing and easy to read. "Type" refers to how the individual characters are designed and arranged.
- **Typeface Definition**: A typeface is the overall design and style of a set of characters, numbers, and symbols. It's like a blueprint for a family of fonts.
- **Font Definition**: A font is a specific variation of a typeface with specific characteristics, such as size, weight, style, and width.
- **Serif Typeface**: This typeface has a classical style with small lines at the end of characters. Serif typefaces are commonly used for printed materials, like books.
- **Sans Serif Typeface**: This typeface has a more modern look, without the small lines at the end of characters. Sans Serif typefaces are commonly used in digital design because they are easy to read on screen. Some examples include Helvetica, Arial, and Roboto.
- **Font Weight**: This is the thickness of the characters, including light, regular, bold, and black.
- **Font Style**: This is the slant and orientation of the characters, like italic and oblique.
- **Font Width**: This is the horizontal space occupied by characters, like condensed and expanded.
- **Baseline**: This is the imaginary line on which most characters rest.
- **Cap Height**: This is the height of uppercase letters, measured from the baseline to the top.
- **X-height**: This is the average height of lowercase letters, excluding ascenders and descenders.
- **Ascenders**: These are the parts of lowercase letters that extend above the x-height, such as the tops of the letters 'h', 'b', 'd', and 'f'.
- **Descenders**: These are the parts of lowercase letters that extend below the baseline, such as the tails of 'y', 'g', 'p', and 'q'.
- **Kerning**: This is how space is adjusted between specific pairs of characters to improve their readability and aesthetics. For example, reducing the space between the letters A and V.
- **Tracking**: This is how space is adjusted between all characters in a block of text. It's essentially the distance between the characters. It affects how dense and open the text will be.
- **Leading**: This is the vertical space between lines of text. It's measured from the baseline of one line to the baseline of the next line.
- **Best Practices with Typography**: You should choose clear and simple fonts to make your designs easy to understand. This is particularly important for the main text of your website. Users are more likely to engage with your content if the font is easy to read. You should use two or three fonts at most to create a visual consistency. Using too many fonts can make the text more difficult to read and weaken your brand's identity. This can also impact the user experience by increasing the load time of the website. You can use font size to create a visual hierarchy for headings, subheadings, paragraphs, and other elements. For example, the main heading on a webpage should have a larger font, followed by subheadings with smaller font sizes. This will give every element in the hierarchy a specific font size that helps users navigate through the structure visually.

CSS `font-family` Property

- **Definition**: A font family is a group of fonts that share a common design. All the fonts that belong to the same family are based on the same core typeface but they also have variations in their style, weight, and width. You can specify multiple font families in order of priority, from highest to lowest, by separating them with commas. These alternative fonts will act as fallback options. You should always include a generic font family at the end of the font-family list.
- **Common Font Families**: Here are some common font families used in CSS: serif, sans-serif, monospace, cursive, fantasy

Web Safe Fonts

- **Definition**: These fonts are a subset of fonts that are very likely to be installed on a computer or device. When the browser has to render a font, it tries to find the font file on the user's system. But if

the font is not found, it will usually fall back to a default system font.

- **Web Safe Fonts**:
    - Sans-serif fonts are commonly used for web development because they don't have small "feet" or lines at the end of the characters, so they're easy to read on screen. Some examples of web-safe sans-serif fonts are: Arial, Verdana, and Trebuchet MS.
    - In contrast, serif fonts do have small "feet" at the end of the characters, so they're commonly used for traditional print. Web safe serif fonts include: Times New Roman and Georgia.

At-rules and the `@font-face` At-rule

- **Definition**: At-rules are statements that provide instructions to the browser. You can use them to define various aspects of the stylesheet, such as media queries, keyframes, font faces, and more.
- **`@font-face`**: This allows you to define a custom font by specifying the font file, format, and other important properties, like weight and style. For the `@font-face` at-rule to be valid, you also need to specify the `src` property. This property contains references to the font resources.
- **Font Formats**: For each font resource, you can also specify the format. This is optional. It's a hint for the browser on the font format. If the format is omitted, the resource will be downloaded and the format will be detected after it's downloaded. If the format is invalid, the resource will not be downloaded. Possible font formats include `collection`, `embedded-opentype`, `opentype`, `svg`, `truetype`, `woff`, and `woff2`.
- **`woff` and `woff2`**: `woff` stands for "Web Open Font Format." It's a widely used font format developed by Mozilla in collaboration with Type Supply, LettError, and other organizations. The difference between `woff` and `woff2` is the algorithm used to compress the data.
- **OpenType**: This is a format for scalable computer fonts developed by Microsoft and Adobe that allows users to access additional features in a font. It's widely used across major operating systems.
- **`tech()`**: This is used to specify the technology of the font resource. This is optional.

Working With External Fonts

- **Definition**: An external font is a font file that is not included directly within your project files. They're usually hosted on a separate server. To include these external fonts inside your project, you can use a `link` element or `@import` statement inside your CSS.
- **Google Fonts**: This is a Google service that offers a collection of fonts, many of which are designed specifically for web development.
- **Font Squirrel**: This is another popular resource that you can use for adding custom external fonts to your projects.

`text-shadow` Property

- **Definition**: This property is used to apply shadows to text. You need to specify the X and Y offset, which represent the horizontal and vertical distance of the shadow from the text, respectively. These values are required. Positive values of the X offset and Y offset will move the shadow right and down, respectively, while negative values will move the shadow left and up.
- **Shadow Color**: You can also customize the color of the shadow by specifying this value before or after the offset. If the color is not specified, the browser will determine the color automatically, so it's usually best to set it explicitly.
- **Blur Radius**: The blur radius is optional but will make the shadow look a lot smoother and more subtle. The default value of the radius blur is zero. The higher the value, the bigger the blur, which means that the shadow becomes lighter.
- **Applying Multiple Text Shadows**: The text can have more than one shadow. The shadows will be applied in layers, from front to back, with the first shadow at the top.

Color Contrast Tools

- **WebAIM's Color Contrast Checker**: This online tool allows you to input the foreground and background colors of your design and instantly see if they meet the Web Content Accessibility Guidelines (WCAG) standards.

- **TPGi Colour Contrast Analyzer**: This is a free color contrast checker that allows you to check if your websites and apps meet the Web Content Accessibility Guidelines (WCAG) standards. This tool also comes with a Color Blindness Simulator feature which allows you to see what your website or app looks like for people with color vision issues.

Accessibility Tree

Accessibility tree is a structure used by assistive technologies, such as screen readers, to interpret and interact with the content on a webpage.

`max()` Function

The `max()` function returns the largest of a set of comma-separated values:

```
img {
  width: max(250px, 25vw);
}
```

In the above example, the width of the image will be 250px if the viewport width is less than 1000 pixels. If the viewport width is greater than 1000 pixels, the width of the image will be 25vw. This is because 25vw is equal to 25% of the viewport width.

Best Practices with CSS and Accessibility

- **`display: none;`**: Using `display: none;` means that screen readers and other assistive technologies won't be able to access this content, as it is not included in the accessibility tree. Therefore, it is important to use this method only when you want to completely remove content from both visual presentation and accessibility.
- **`visibility: hidden;`**: This property and value hides the content visually but keeps it in the document flow, meaning it still occupies space on the page. These elements will also no longer be read by screen readers because they will have been removed from the accessibility tree.
- **`.sr-only` CSS class**: This is a common technique used to visually hide content while keeping it accessible to screen readers.

```
.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  padding: 0;
  margin: -1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  white-space: nowrap;
  border: 0;
}
```

- **`scroll-behavior: smooth;`**: This property and value enables a smooth scrolling behavior.
- **`prefers-reduced-motion` feature**: This is a media feature that can be used to detect the user's animation preference.

```
@media (prefers-reduced-motion: no-preference) {
  * {
    scroll-behavior: smooth;
  }
}
```

In the above example, smooth scrolling is enabled if the user doesn't have animation preference set on their device.

Hiding Content with HTML Attributes

- **`aria-hidden` attribute**: Used to hide an element from people using assistive technology such as screen readers. For example, this can be used to hide decorative images that do not provide any meaningful content.
- **`hidden` attribute**: This attribute is supported by most modern browsers and hides content both visually and from the accessibility tree. It can be easily toggled with JavaScript.

```
<p aria-hidden>This paragraph is visible for sighted users, but is hidden
from assistive technology.</p>
<p hidden>This paragraph is hidden from both sighted users and assistive
technology.</p>
```

Accessibility Issue of the `placeholder` Attribute

Using placeholder text is not great for accessibility. Too often, users confuse the placeholder text with an actual input value - they think there is already a value in the input.

Working with Different Attribute Selectors and Links

- **Definition**: The `attribute` selector allows you to target HTML elements based on their attributes like the `href` or `title` attributes.
- **`title` Attribute**: This attribute provides additional information about an element.

Targeting Elements with the `lang` and `data-lang` Attribute

- **`lang` Attribute**: This attribute is used in HTML to specify the language of the content within an element. You might want to style elements differently based on the language they are written in, especially on a multilingual website.
- **`data-lang` Attribute**: Custom data attributes like the `data-lang` attribute are commonly used to store additional information in elements, such as specifying the language used within a specific section of text.

Working with the Attribute Selector, Ordered List Elements and the `type` Attribute

- **`type` Attribute**: When working with ordered lists in HTML, the `type` attribute allows you to specify the style of numbering used, such as numerical, alphabetical, or Roman numerals.

Working With Floats

- **Definition**: Floats are used to remove an element from its normal flow on the page and position it either on the left or right side of its container. When this happens, text will wrap around that floated content.
- **Clearing Floats**: The `clear` property is used to determine if an element needs to be moved below the floated content. When you have multiple floated elements stacked next to each other, there could be issues with overlap and collapsing in the layouts. So a `clearfix` hack was created to fix this issue.

Static, Relative and Absolute Positioning

- **Static Positioning**: This is the normal flow for the document. Elements are positioned from top to bottom, and left to right one after another.
- **Relative Positioning**: This allows you to use the `top`, `left`, `right` and `bottom` properties to position the element within the normal document flow. You can also use relative positioning to make

elements overlap with other elements on the page.
- **Absolute Positioning**: This allows you to take an element out of the normal document flow, making it behave independently from other elements.

Fixed and Sticky Positioning

- **Fixed Positioning**: When an element is positioned with `position: fixed`, it is removed from the normal document flow and placed relative to the viewport, meaning it stays in the same position even when the user scrolls. This is often used for elements like headers or navigation bars that need to remain visible at all times.
- **Sticky Positioning**: This type of positioning will act as an relative positioned element as you scroll down the page. If you specify a `top`, `left`, `right` or `bottom` property, then the element will stop acting like a relatively positioned element and start behaving like a fixed position element.

Working With the `z-index` Property

- **Definition**: The `z-index` property in CSS is used to control the vertical stacking order of positioned elements that overlap on the page.

Responsive Web Design

- **Definition**: The core principle of responsive design is adaptability – the ability of a website to adjust its layout and content based on the screen size and capabilities of the device it's being viewed on.
- **Fluid grids**: These use relative units like percentages instead of fixed units like pixels, allowing content to resize and reflow based on screen size.
- **Flexible images**: These are set to resize within their containing elements, ensuring they don't overflow their containers on smaller screens.

Media Queries

- **Definition**: This allows developers to apply different styles based on the characteristics of the device, primarily the viewport width.
- **`all` Media Type**: This is suitable for all devices. This is the default if no media type is specified.
- **`print` Media Types**: This is intended for paged material and documents viewed on a screen in print preview mode.
- **`screen` Media Types**: This is intended primarily for screens.
- **`aspect-ratio`**: This describes the ratio between the width and height of the viewport.
- **`orientation`**: This is used to indicate whether the device is in landscape or portrait orientation.
- **`resolution`**: This is used to describe the resolution of the output device in dots per inch (dpi) or dots per centimeter (dpcm).
- **`hover`**: This is used to test whether the primary input mechanism can hover over elements.
- **`prefers-color-scheme`**: This is used to detect if the user has requested a light or dark color theme.
- **Media Queries and Logical Operators**: The `and` operator is used to combine multiple media features, while `not` and `only` can be used to negate or isolate media queries.

Common Media Breakpoints

- **Definition**: Media breakpoints are specific points in a website's design where the layout and content adjust to accommodate different screen sizes. There are some general breakpoints that you can use to target phones, tablets and desktop screens. But it is not wise to try to chase down every single possible screen size for different devices.
- **Small Devices (smartphones)**: up to 640px
- **Medium Devices (tablets)**: 641px to 1024px
- **Large Devices (desktops)**: 1025px and larger

Mobile first approach

- **Definition**: The `mobile-first` approach is a design philosophy and development strategy in responsive web design that prioritizes creating websites for mobile devices before designing for larger screens.

CSS Custom Properties (CSS Variables)

- **Definition**: CSS custom properties, also known as CSS variables, are entities defined by CSS authors that contain specific values to be reused throughout a document. They are a powerful feature that allows for more efficient, maintainable, and flexible stylesheets. Custom properties are particularly useful in creating themeable designs. You can define a set of properties for different themes.

The `@property` Rule

- **Definition**: The `@property` rule is a powerful CSS feature that allows developers to define custom properties with greater control over their behavior, including how they animate and their initial values.

```
@property --property-name {
  syntax: '<type>';
  inherits: true | false;
  initial-value: <value>;
}
```

- **`--property-name`**: This is the name of the custom property you're defining. Like all custom properties, it must start with two dashes. `--property-name` can be things like `<color>`, `<length>`, `<number>`, `<percentage>`, or more complex types.
- **`syntax`**: This defines the type of the property.
- **`inherits`**: This specifies whether the property should inherit its value from its parent element.
- **`initial-value`**: This sets the default value of the property.
- **Fallbacks**: When using the custom property, you can provide a fallback value using the `var()` function, just as you would with standard custom properties.

CSS Grid Basics

- **Definition**: CSS Grid is a two-dimensional layout system used to create complex layouts in web pages. Grids will have rows and columns with gaps between them. To define a grid layout, you would set the `display` property to `grid`.
- **The `fr` (Fractional) Unit**: This unit represents a fraction of the space within the grid container. You can use the `fr` unit to create flexible grids.
- **Creating Gaps Between Tracks**: There are three ways to create gaps between tracks in CSS grid. You can use the `column-gap` property to create gaps between columns. You can use the `row-gap` property to create gaps between rows. Or you can use the `gap` shorthand property to create gaps between both rows and columns.
- **`repeat()` Function**: This function is used to repeat sections in the track listing. Instead of writing `grid-template-columns: 1fr 1fr 1fr;` you can use the `repeat()` function instead.
- **Explicit Grid**: You can specify the number of lines or tracks using the `grid-template-columns` or `grid-template-rows` properties.
- **Implicit Grid**: When items are placed outside of the grid, then rows and columns are automatically created for those outside elements.
- **`minmax()` Function**: This function is used to set the minimum and maximum sizes for a track.
- **Line-based Placement**: All grids have lines. To specify where the item begins on a line, you can use the `grid-column-start` and `grid-row-start` properties. To specify where the item ends on the line, you can use the `grid-column-end` and `grid-row-end` properties. You can also choose to use the `grid-column` or `grid-row` shorthand properties instead.
- **`grid-template-areas`**: This property is used to provide a name for the items you are going to position on the grid.

CSS Animation Basics

- **Definition**: CSS animations allow you to create dynamic, visually engaging effects on web pages without the need for JavaScript or complex programming. They provide a way to smoothly transition elements between different styles over a specified duration.
- **The `@keyframes` Rule**: This rule defines the stages and styles of the animation. It specifies what styles the element should have at various points during the animation.
- **`animation` Property**: This is the shorthand property used to apply animations.
- **`animation-name`**: This specifies the name for the `@keyframes` rule to use.
- **`animation-duration`**: This sets how long the animation should take to complete.
- **`animation-timing-function`**: This defines how the animation progresses over time (such as ease, linear, ease-in-out).
- **`animation-delay`**: This specifies a delay before the animation starts.
- **`animation-iteration-count`**: This sets how many times the animation should repeat.
- **`animation-direction`**: This determines whether the animation should play forwards, backwards, or alternate.
- **`animation-fill-mode`**: This specifies how the element should be styled before and after the animation.
- **`animation-play-state`**: This allows you to pause and resume the animation.

Accessibility and the `prefers-reduced-motion` Media Query

- **The `prefers-reduced-motion` Media Query**: One of the primary accessibility concerns with animations is that they can cause discomfort or even physical harm to some users. People with vestibular disorders or motion sensitivity may experience dizziness, nausea, or headaches when exposed to certain types of movement on screen. The `prefers-reduced-motion` media query allows web developers to detect if the user has requested minimal animations or motion effects at the system level.

--assignment--

Review the CSS topics and concepts.

CSS Accessibility Review

Color Contrast Tools

- **WebAIM's Color Contrast Checker**: This online tool allows you to input the foreground and background colors of your design and instantly see if they meet the Web Content Accessibility Guidelines (WCAG) standards.
- **TPGi Colour Contrast Analyzer**: This is a free color contrast checker that allows you to check if your websites and apps meet the Web Content Accessibility Guidelines (WCAG) standards. This tool also comes with a Color Blindness Simulator feature which allows you to see what your website or app looks like for people with color vision issues.

Accessibility Tree

Accessibility tree is a structure used by assistive technologies, such as screen readers, to interpret and interact with the content on a webpage.

`max()` Function

The `max()` function returns the largest of a set of comma-separated values:

```
```

```
img {
  width: max(250px, 25vw);
}
```

In the above example, the width of the image will be 250px if the viewport width is less than 1000 pixels. If the viewport width is greater than 1000 pixels, the width of the image will be 25vw. This is because 25vw is equal to 25% of the viewport width.

Best Practices with CSS and Accessibility

- `display: none;`: Using `display: none;` means that screen readers and other assistive technologies won't be able to access this content, as it is not included in the accessibility tree. Therefore, it is important to use this method only when you want to completely remove content from both visual presentation and accessibility.
- `visibility: hidden;`: This property and value hides the content visually but keeps it in the document flow, meaning it still occupies space on the page. These elements will also no longer be read by screen readers because they will have been removed from the accessibility tree.
- `.sr-only` **CSS class**: This is a common technique used to visually hide content while keeping it accessible to screen readers.

```
.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  padding: 0;
  margin: -1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  white-space: nowrap;
  border: 0;
}
```

- `scroll-behavior: smooth;`: This property and value enables a smooth scrolling behavior.
- `prefers-reduced-motion` **feature**: This is a media feature that can be used to detect the user's animation preference.

```
@media (prefers-reduced-motion: no-preference) {
  * {
    scroll-behavior: smooth;
  }
}
```

In the above example, smooth scrolling is enabled if the user doesn't have animation preference set on their device.

Hiding Content with HTML Attributes

- `aria-hidden` **attribute**: Used to hide an element from people using assistive technology such as screen readers. For example, this can be used to hide decorative images that do not provide any meaningful content.
- `hidden` **attribute**: This attribute is supported by most modern browsers and hides content both visually and from the accessibility tree. It can be easily toggled with JavaScript.

```
<p aria-hidden>This paragraph is visible for sighted users, but is hidden
from assistive technology.</p>
```

```
<p hidden>This paragraph is hidden from both sighted users and assistive
technology.</p>
```

Accessibility Issue of the `placeholder` Attribute

Using placeholder text is not great for accessibility. Too often, users confuse the placeholder text with an actual input value - they think there is already a value in the input.

--assignment--

Review the CSS Accessibility topics and concepts.

CSS Animations Review

CSS Animation Basics

- **Definition**: CSS animations allow you to create dynamic, visually engaging effects on web pages without the need for JavaScript or complex programming. They provide a way to smoothly transition elements between different styles over a specified duration.
- **The `@keyframes` Rule**: This rule defines the stages and styles of the animation. It specifies what styles the element should have at various points during the animation.

```
@keyframes slide-in {
  0% {
    transform: translateX(-100%);
  }
  100% {
    transform: translateX(0);
  }
}
```

- **`animation` Property**: This is the shorthand property used to apply animations.
- **`animation-name`**: This specifies the name for the `@keyframes` rule to use.
- **`animation-duration`**: This sets how long the animation should take to complete.
- **`animation-timing-function`**: This defines how the animation progresses over time (such as ease, linear, ease-in-out).
- **`animation-delay`**: This specifies a delay before the animation starts.
- **`animation-iteration-count`**: This sets how many times the animation should repeat.
- **`animation-direction`**: This determines whether the animation should play forwards, backwards, or alternate.
- **`animation-fill-mode`**: This specifies how the element should be styled before and after the animation.
- **`animation-play-state`**: This allows you to pause and resume the animation.

Accessibility and the `prefers-reduced-motion` Media Query

- **The `prefers-reduced-motion` Media Query**: One of the primary accessibility concerns with animations is that they can cause discomfort or even physical harm to some users. People with vestibular disorders or motion sensitivity may experience dizziness, nausea, or headaches when exposed to certain types of movement on screen. The `prefers-reduced-motion` media query allows web developers to detect if the user has requested minimal animations or motion effects at the system level.

```
```

```
.animated-element {
  transition: transform 0.3s ease-in-out;
}

@media (prefers-reduced-motion: reduce) {
  .animated-element {
    transition: none;
  }
}
```

--assignment--

Review the CSS Animations topics and concepts.

CSS Attribute Selectors Review

Working with Different Attribute Selectors and Links

- **Definition**: The `attribute` selector allows you to target HTML elements based on their attributes like the `href` or `title` attributes.

```
a[href] {
  color: blue;
  text-decoration: underline;
}
```

- **`title` Attribute**: This attribute provides additional information about an element. Here is how you can target links with the `title` attribute:

```
a[title] {
  font-weight: bold;
  text-decoration: none;
}
```

Targeting Elements with the `lang` and `data-lang` Attribute

- **`lang` Attribute**: This attribute is used in HTML to specify the language of the content within an element. You might want to style elements differently based on the language they are written in, especially on a multilingual website.

```
p[lang="en"] {
  font-style: italic;
}
```

- **`data-lang` Attribute**: Custom data attributes like the `data-lang` attribute are commonly used to store additional information in elements, such as specifying the language used within a specific section of text. Here is how you can style elements based on the `data-lang` attribute:

```
div[data-lang="fr"] {
  color: blue;
}
```

Working with the Attribute Selector, Ordered List Elements and the `type` Attribute

- **`type` Attribute**: When working with ordered lists in HTML, the `type` attribute allows you to specify the style of numbering used, such as numerical, alphabetical, or Roman numerals.

```
/*Example targeting uppercase alphabetical numbering*/
ol[type="A"] {
  color: purple;
  font-weight: bold;
}

/*Example targeting lowercase Roman numerals*/
ol[type="i"] {
  color: green;
}
```

--assignment--

Review the CSS Attribute Selectors topics and concepts.

Lists, Links, CSS Background and Borders Review

Styling Lists

- **`line-height` Property**: This property is used to create space between lines of text. The accepted `line-height` values include the keyword `normal`, numbers, percentages and length units like the `em` unit.
- **`list-style-type` Property**: This property is used to specify the marker for a list item. Acceptable values can include a circle, disc, or decimal.
- **`list-style-position` Property**: This property is used to set the position for the list marker. The only two acceptable values are inside and outside.
- **`list-style-image` Property**: This property is used to use an image for the list item marker. A common use case is to use the `url` function with a value set to a valid image location.

Spacing list items using `margin`

- Apart from `line-height`, margins can also be used in CSS to enhance the spacing and readability of list items.
- Margins create space outside each `li` element, allowing control over the gap between list items.
- `margin-bottom` is used to create space below each list item. For example, `margin-bottom: 10px;` will create a 10-pixel gap below each list item.

Styling Links

- **`pseudo-class`**: This is a keyword added to a selector that allows you to select elements based on a particular state. Common states would include the `:hover`, `:visited` and `:focus` states.
- **`:link pseudo-class`**: This pseudo-class is used to style links that have not be visited by the user.
- **`:visited pseudo-class`**: This pseudo-class is used to style links where a user has already visited.
- **`:hover pseudo-class`**: This pseudo-class is used to style an elements where a user is actively hovering over them.
- **`:focus pseudo-class`**: This pseudo-class is used to style an element when it receives focus. Examples would include `input` or `select` elements where the clicks or tabs on the element to focus

it.
- `:active pseudo-class`: This pseudo-class is used to style an element that was activated by the user. A common example would be when the user clicks on a button.

Working with Backgrounds and Borders

- `background-size` **Property**: This property is used to set the background size for an element. Some common values include `cover` for the background image to cover the entire element and `contain` for the background image to fit within the element.
- `background-repeat` **Property**: This property is used to determine how background images should be repeated along the horizontal and vertical axes. The default value for `background-repeat` is `repeat` meaning the image will repeat both horizontally and vertically. You can also specify that there should be no repeat by using the `no-repeat` property.
- `background-position` **Property**: This property is used to specify the position of the background image. It can be set to a specific length, percentage, or keyword values like `top`, `bottom`, `left`, `right`, and `center`.
- `background-attachment` **Property**: This property is used to specify whether the background image should scroll with the content or remain fixed in place. The main values are `scroll` (default), where the background image scrolls with the content, and `fixed`, where the background image stays in the same position on the screen.
- `background-image` **Property**: This property is used to set the background image of an element. You can set multiple background images at the same time and use either the `url`, `radial-gradient` or `linear-gradient` functions as values.
- `background` **Property**: This is the shorthand property for setting all background properties in one declaration. Here is an example of setting the background image and setting it to not repeat: `background: no-repeat url("example-url-goes-here");`
- **Good Contrast for Background and Foreground Colors**: It is important to ensure that the background and foreground colors have good contrast to make the text readable. The Web Content Accessibility Guidelines (WCAG) recommend a minimum contrast ratio of 4.5:1 for normal text and 3:1 for large text.

Borders

- `border-top` **Property**: This property is used to set the styles for the top border of an element. `border-top: 3px solid blue;` sets a 3-pixel-wide solid blue border on the top side of the element.
- `border-right` **Property**: This property is used to set the styles for the right border of an element. `border-right: 2px solid red;` sets a 2-pixel-wide solid red border on the right side of the element.
- `border-bottom` **Property**: This property is used to set the styles for the bottom border of an element. `border-bottom: 1px dashed green;` sets a 1-pixel-wide dashed green border on the bottom side of the element.
- `border-left` **Property**: This property is used to set the styles for the left border of an element. `border-left: 4px dotted orange;` sets a 4-pixel-wide dotted orange border on the left side of the element.
- `border` **Property**: This is the shorthand property for setting the width, style, and color of an element's border. `border: 1px solid black;` sets a 1-pixel-wide solid black border.
- `border-radius` **Property**: This property is used to create rounded corners for an element's border.
- `border-style` **Property**: This property is used to set the style of an element's border. Some accepted values include `solid`, `dashed`, `dotted`, and `double`.

Gradients

- `linear-gradient()` **Function**: This CSS function is used to create a transition between multiple colors along a straight line.
- `radial-gradient()` **Function**: This CSS function creates an image that radiates from a particular point, like a circle or an ellipse, and gradually transitions between multiple colors.

--assignment--

Review the CSS Backgrounds and Borders topics and concepts.

CSS Colors Review

Color Theory

- **Color Theory Definition**: This is the study of how colors interact with each other and how they affect our perception. It covers color relationships, color harmony, and the psychological impact of color.
- **Primary Colors**: These colors which are yellow, blue, and red, are the fundamental hues from which all other colors are derived.
- **Secondary Colors**: These colors result from mixing equal amounts of two primary colors. Green, orange, and purple are examples of secondary colors.
- **Tertiary Colors**: These colors result from combining a primary color with a neighboring secondary color. Yellow-Green, Blue-Green, and Blue-Violet are examples of tertiary colors.
- **Warm Colors**: These colors which include reds, oranges, and yellows, evoke feelings of comfort, warmth, and coziness.
- **Cool Colors**: These colors which include blues, green, and purples, evoke feelings of calmness, serenity, and professionalism.
- **Color Wheel**: The color wheel is a circular diagram that shows how colors relate to each other. It's an essential tool for designers because it helps them to select color combinations.
- **Analogous Color Schemes**: These color schemes create cohesive and soothing experiences. They have analogous colors, which are adjacent to each other in the color wheel.
- **Complementary Color Schemes**: These color schemes create high contrast and visual impact. Their colors are located on the opposite ends of the color wheel, relative to each other.
- **Triadic Color Scheme**: This color scheme has vibrant colors. They are made from colors that are approximately equidistant from each other. If they are connected, they form an equilateral triangle on the color wheel.
- **Monochromatic Color Scheme**: For this color scheme, all the colors are derived from the same base color by adjusting its lightness, darkness, and saturation. This evokes a feeling of unity and harmony while still creating contrast.

Different Ways to Work with Colors in CSS

- **Named Colors**: These colors are predefined color names recognized by browsers. Examples include `blue`, `darkred`, `lightgreen`.
- **`rgb()` Function**: RGB stands for Red, Green, and Blue — the primary colors of light. These three colors are combined in different intensities to create a wide range of colors. the `rgb()` function allows you to define colors using the RGB color model.

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

- **`rgba()` Function**: This function adds a fourth value, alpha, that controls the transparency of the color. If not provided, the alpha value defaults to `1`.

```
div {
  background-color: rgba(0, 0, 255, 0.5);
}
```

- `hsl()` **Function**: HSL stands for Hue, Saturation, and Lightness — three key components that define a color.

```
p {
  color: hsl(120, 100%, 50%);
}
```

- `hsla()` **Function**: This function adds a fourth value, alpha, that controls the opacity of the color.

```
div {
  background-color: hsla(0, 100%, 50%, 0.5);
}
```

- **Hexadecimal**: A hex code (short for hexadecimal code) is a six-character string used to represent colors in the RGB color model. The "hex" refers to the base-16 numbering system, which uses digits 0 to 9 and letters A to F.

```
h1 {
  color: #FF5733; /* A reddish-orange color */
}

p {
  background-color: #4CAF50; /* A shade of green */
}
```

Linear and Radial Gradients

- **Linear Gradients**: These gradients create a gradual blend between colors along a straight line. You can control the direction of this line using keywords like `to top`, `to right`, `to bottom right`, or angles like `45deg`. You can use any valid CSS color and as many color stops as you would like.

```
.linear-gradient {
  background: linear-gradient(45deg, red, #33FF11, rgba(100, 100, 255, 0.5));
  height: 40vh;
}
```

- **Radial Gradients**: These gradients create circular or elliptical gradients that radiate from a central point.

```
.radial-gradient {
  background: radial-gradient(circle, red, blue);
  height: 40vh;
}
```

--assignment--

Review the CSS Colors topics and concepts.

CSS Flexbox Review

Introduction to CSS Flexbox and Flex Model

- **Definition**: CSS flexbox is a one-dimensional layout model that allows you to arrange elements in rows and columns within a container.
- **Flex Model**: This model defines how flex items are arranged within a flex container. Every flex container has two axes: the main axis and the cross axis.

The `flex-direction` Property

- **Definition**: This property sets the direction of the main axis. The default value of `flex-direction` is `row`, which places all the flex items on the same row, in the direction of your browser's default language (left to right or right to left).
- **`flex-direction: row-reverse;`**: This reverses the items in the row.
- **`flex-direction: column;`**: This will align the flex items vertically instead of horizontally.
- **`flex-direction: column-reverse;`**: This reverses the order of the flex items vertically.

The `flex-wrap` Property

- **Definition**: This property determines how flex items are wrapped within a flex container to fit the available space. `flex-wrap` can take three possible values: `nowrap`, `wrap`, and `wrap-reverse`.
- **`flex-wrap: nowrap;`**: This is the default value. Flex items won't be wrapped onto a new line, even if their width exceed the container's width.
- **`flex-wrap: wrap;`**: This property will wrap the items when they exceed the width of their container.
- **`flex-wrap: wrap-reverse;`**: This property will wrap flex items in reverse order.
- **`flex-flow` Property**: This property is a shorthand property for `flex-direction` and `flex-wrap`.

```
flex-flow: column wrap-reverse;
```

The `justify-content` Property

- **Definition**: This property aligns the child elements along the main axis of the flex container.
- **`justify-content: flex-start;`**: In this case, the flex items will be aligned to the start of the main axis. This could be horizontal or vertical.
- **`justify-content: flex-end;`**: In this case, the flex items are aligned to the end of the main axis, horizontally or vertically.
- **`justify-content: center;`**: This centers the flex items along the main axis.
- **`justify-content: space-between;`**: This will distribute the elements evenly along the main axis.
- **`justify-content: space-around;`**: This will distribute flex items evenly within the main axis, adding a space before the first item and after the last item.
- **`justify-content: space-evenly;`**: This will distribute the items evenly along the main axis.

The `align-items` Property

- **Definition**: This property is used to distribute items along the cross axis. Remember that the cross axis is perpendicular to the main axis.
- **`align-items: center;`**: This is used to center the items along the cross axis.
- **`align-items: flex-start;`**: This aligns the items to the start of the cross axis.
- **`align-items: stretch;`**: This is used to stretch the flex items along the cross axis.

--assignment--

Review the CSS Flexbox topics and concepts.

CSS Grid Review

CSS Grid Basics

- **Definition**: CSS Grid is a two-dimensional layout system used to create complex layouts in web pages. Grids will have rows and columns with gaps between them. To define a grid layout, you would set the `display` property to `grid`.

```
.container {
  display: grid;
}
```

- **The `fr` (Fractional) Unit**: This unit represents a fraction of the space within the grid container. You can use the `fr` unit to create flexible grids.
- **Creating Gaps Between Tracks**: There are three ways to create gaps between tracks in CSS grid. You can use the `column-gap` property to create gaps between columns. You can use the `row-gap` property to create gaps between rows. Or you can use the `gap` shorthand property to create gaps between both rows and columns.
- **`grid-template-columns`**: This is used to set lines names and sizing for the grid track columns.

```
.container {
  display: grid;
  width: 100%;
  grid-template-columns: 30px 1fr;
}
```

- **`grid-template-rows`**: This is used to set lines names and sizing for the grid track rows.
- **`grid-auto-flow`**: The determines how auto placed items fit in the grid.

```
.container {
  display: grid;
  width: 100%;
  grid-auto-flow: column;
}
```

- **`grid-auto-columns`**: This is used to set the size for columns created implicitly.

```
.container {
  display: grid;
  width: 100%;
  grid-auto-columns: auto;
}
```

- **`place-items`**: This is used to align items for both block and inline directions.
- **`align-items`**: This is used to set the alignment for the items in a grid container.
- **`repeat()` Function**: This function is used to repeat sections in the track listing. Instead of writing `grid-template-columns: 1fr 1fr 1fr;` you can use the `repeat()` function instead.

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
}
```

- **Explicit Grid**: You can specify the number of lines or tracks using the `grid-template-columns` or `grid-template-rows` properties.
- **Implicit Grid**: When items are placed outside of the grid, then rows and columns are automatically created for those outside elements.
- **`minmax()` Function**: This function is used to set the minimum and maximum sizes for a track.

```
.container {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  grid-auto-rows: minmax(150px, auto);
}
```

- **Line-based Placement**: All grids have lines. To specify where the item begins on a line, you can use the `grid-column-start` and `grid-row-start` properties. To specify where the item ends on the line, you can use the `grid-column-end` and `grid-row-end` properties. You can also choose to use the `grid-column` or `grid-row` shorthand properties instead.

Here is an example of using the `grid-column` property to make an element stretch across all columns.

```
.element {
  grid-column: 1 / -1;
}
```

- **`grid-template-areas`**: The property is used to provide a name for the items you are going to position on the grid.

```
<div class="container">
  <div class="header">Header</div>
  <div class="sidebar">Sidebar</div>
  <div class="main">Main Content</div>
  <div class="footer">Footer</div>
</div>
```

```
.container {
  display: grid;
  grid-template-columns: 200px 1fr;
  grid-template-rows: auto 1fr auto;
  grid-template-areas:
    "header header"
    "sidebar main"
    "footer footer";
  gap: 20px;
}

.header {
  grid-area: header;
  background-color: #4CAF50;
  padding: 10px;
  color: white;
}

.sidebar {
  grid-area: sidebar;
  background-color: #f4f4f4;
  padding: 10px;
```

```
}

.main {
  grid-area: main;
  background-color: #e0e0e0;
  padding: 10px;
}

.footer {
  grid-area: footer;
  background-color: #4CAF50;
  padding: 10px;
  color: white;
}
```

Debugging CSS

- **DevTools (Developer Tools)**: DevTools allow you to inspect and modify your CSS in real-time. The Styles pane shows all the CSS rules applied to the selected element, including inherited styles. You can toggle individual properties on and off, edit values, and even add new rules directly in the browser. This immediate feedback is incredibly useful for experimenting with different styles without modifying your source code.
- **CSS Validators**: Validators are used to check your CSS against the official specifications and reports any errors or warnings. A popular validator you can use is the W3C CSS Validator.
- **Debugging Responsive Designs**: The DevTools has an option to allow you to simulate how your site looks on various screen sizes and devices. This can help you identify breakpoint issues or styles that don't scale well across different viewport sizes.

--assignment--

Review the CSS Grid topics and concepts.

CSS Layouts and Effects Review

CSS Overflow Property

- **Definition**: Overflow refers to the way elements handle content that exceeds, or "overflows", the size of the containing element. Overflow is two-dimensional.
- `overflow-x`: The x-axis determines horizontal overflow.
- `overflow-y`: the y-axis determines vertical overflow.
- `overflow`: Shorthand property for `overflow-x` and `overflow-y`. If given one value, both overflows will use it. If given two values, the `overflow-x` will use the first, and the `overflow-y` will use the second.

CSS Transform Property

- **Definition**: This property enables you to apply various transformations to elements, such as rotating, scaling, skewing, or translating (moving) them in 2D or 3D space.
- `translate()` **Function**: This function is used to move an element from its current position.
- `scale()` **Function**: This function allows you to change the size of an element.
- `rotate()` **Function**: This function allows you to rotate an element.
- `skew()` **Function**: This function allows you to skew an element.
- **Transforms and Accessibility**: If you're using transform to hide or reveal content, make sure that the content is still accessible to screen readers and keyboard navigation. Hidden content should be truly hidden, such as by using `display: none` or `visibility: hidden`, rather than simply being visually moved off-screen.

The Box Model

- **Definition**: In the CSS box model, every element is surrounded by a box. This box consists of four components: the content area, `padding`, `border`, `margin`.
- **Content Area**: The content area is the innermost part of the box. It's the space that contains the actual content of an element, like text or images.
- **`padding`**: The padding is the area immediately after the content area. It's the space between the content area and the border of an element.
- **`border`**: The border is the outer edge or outline of an element in the CSS box model. It's the visual boundary of the element.
- **`margin`**: The margin is the space outside the border of an element. It determines the distance between an element and other elements around it.

Margin Collapse

- **Definition**: This behavior occurs when the vertical margins of adjacent elements overlap, resulting in a single margin equal to the larger of the two. This behavior applies only to vertical margins (top and bottom), not horizontal margins (left and right).

The `content-box` and `border-box` Property Values

- **`box-sizing` Property**: This property is used to determine how the final `width` and `height` are calculated for an HTML element.
- **`content-box` Value**: In the `content-box` model, the `width` and `height` that you set for an element determine the dimensions of the content area but they don't include the `padding`, `border`, or `margin`.
- **`border-box` Value**: With `border-box`, the `width` and `height` of an element include the content area, the `padding`, and the `border`, but they don't include the `margin`.

CSS Reset

- **Definition**: A CSS reset is a stylesheet that removes all or some of the default formatting that web browsers apply to HTML elements. Third party options for CSS resets include `sanitize.css` and `normalize.css`.

CSS Filter Property

- **Definition**: This property can be used to create various effects such as blurring, color shifting, and contrast adjustment.
- **`blur()` Function**: This function applies a Gaussian blur to the element. The amount is defined in pixels and represents the radius of the blur.
- **`brightness()` Function**: This function adjusts the brightness of the element. A value of 0% will make the element completely black, while values over 100% will increase the brightness.
- **`contrast()` Function**: This function adjusts the contrast of the element. A value of 0% will make the element completely grey, 100% will make the element appear normally, and values greater than 100% will increase the contrast.
- **`grayscale()` Function**: This function converts the element to grayscale. The amount is defined as a percentage, where 100% is completely grayscale and 0% leaves the image unchanged.
- **`sepia()` Function**: This function applies a sepia tone to the element. Like grayscale, it uses a percentage value.
- **`hue-rotate()` Function**: This function applies a hue rotation to the element. The value is defined in degrees and represents a rotation around the color circle.

--assignment--

Review the CSS Layouts and Effects topics and concepts.

CSS Libraries and Frameworks Review

CSS Frameworks

- **CSS frameworks**: CSS frameworks can speed up your workflow, create a uniform visual style across a website, make your design look consistent across multiple browsers, and keep your CSS code more organized.
- **Popular CSS frameworks**: Some of the popular CSS frameworks are Tailwind CSS, Bootstrap, Materialize, and Foundation.
- **Potential disadvantages**:
  - The CSS provided by the framework might conflict with your custom CSS.
  - Your website might look similar to other websites using the same framework.
  - Large frameworks might cause performance issues.

Two Types of CSS Frameworks

- **Utility-first CSS frameworks**: These frameworks have small classes with specific purposes, like setting the margin, padding, or background color. You can assign these small classes directly to the HTML elements as needed. Tailwind CSS is categorized as a utility-first CSS framework.

Here is an example of using Tailwind CSS to style a button.

```
<button class="bg-blue-500 text-white font-bold py-2 px-4 rounded-full
hover:bg-blue-700">
  Button
</button>
```

- **Component-based CSS frameworks**: These frameworks have pre-built components with pre-defined styles that you can add to your website. The components are available in the official documentation of the CSS framework, and you can copy and paste them into your project. Bootstrap is categorized as a component-based CSS framework.

Here is an example of using Bootstrap to create a list group. Instead of applying small classes to your HTML elements, you will add the entire component, including the HTML structure.

```
<div class="card" style="width: 25rem;">
  <ul class="list-group list-group-flush">
    <li class="list-group-item">HTML</li>
    <li class="list-group-item">CSS</li>
    <li class="list-group-item">JavaScript</li>
  </ul>
</div>
```

Tailwind CSS

Tailwind is a utility-first CSS framework. Instead of writing custom CSS rules, you build your designs by combining small utility classes directly in your HTML.

Responsive Design Utilities

Tailwind uses prefixes such as `sm:`, `md:`, and `lg:` to apply styles at different screen sizes.

```
<div class="w-full md:w-1/2 lg:flex-row">Responsive layout</div>
```

Flexbox Utilities

Classes like `flex`, `flex-col`, `justify-around`, and `items-center` make it easy to create flexible layouts.

```
<div class="flex flex-col md:flex-row justify-around items-center">
  <p>Column on small screens</p>
  <p>Row on medium and larger screens</p>
</div>
```

Grid Utilities

Tailwind includes utilities for CSS Grid, like `grid`, `grid-cols-1`, and `md:grid-cols-3`.

```
<div class="grid grid-cols-1 md:grid-cols-3 gap-8">
  <div class="bg-gray-100 p-4">Column 1</div>
  <div class="bg-gray-100 p-4">Column 2</div>
  <div class="bg-gray-100 p-4">Column 3</div>
</div>
```

Spacing Utilities

Utilities like `mt-8`, `mx-auto`, `p-4`, and `gap-4` help create consistent spacing without writing CSS.

```
<div class="mt-8 p-4 bg-indigo-600 text-white">Spaced content</div>
```

Typography Utilities

Utilities like `uppercase`, `font-bold`, `font-semibold`, and `text-4xl` control text appearance.

You can set font sizes that adjust at breakpoints, such as `text-3xl md:text-5xl`.

```
<h1 class="text-3xl md:text-5xl font-semibold text-center">Responsive Heading
</h1>
```

Colors and Hover States

Tailwind provides a wide color palette, such as `text-red-700`, `bg-indigo-600`, and `bg-gray-100`.

Classes like `hover:bg-pink-600` make interactive effects simple.

```
<a href="#" class="bg-pink-500 hover:bg-pink-600 text-white px-4 py-2
rounded-md">
  Hover Me
</a>
```

Borders, Rings, and Effects

- **Borders**: `border-2 border-red-300` adds borders with specified thickness and colors.
- **Rings**: `ring-1 ring-gray-300` creates outline-like effects often used for focus or cards.
- **Rounded corners and scaling**: Classes like `rounded-md`, `rounded-xl`, and `scale-105` add polish.

```
<div class="p-6 rounded-xl ring-2 ring-fuchsia-500 scale-105">
  Highlighted card
</div>
```

Gradients

Tailwind supports gradient utilities like `bg-gradient-to-r from-fuchsia-500 to-indigo-600`.

```
<div class="p-4 text-white bg-gradient-to-r from-fuchsia-500 to-indigo-600">
  Gradient background
</div>
```

CSS Preprocessors

- **CSS preprocessor**: A CSS preprocessor is a tool that extends standard CSS. It compiles the code with extended syntax into a native CSS file. It can be helpful for writing cleaner, reusable, less repetitive, and scalable CSS for complex projects.
- **Features**: Some of the features that can be provided by CSS preprocessors are variables, mixins, nesting, and selector inheritance.
- **Popular CSS preprocessors**: Some of the popular CSS preprocessors are Sass, Less, and Stylus.
- **Potential disadvantages**:
  - Compiling the CSS rules into standard CSS might cause overhead.
  - The compiled code may be difficult to debug.

Sass

- **Sass**: It is one of the most popular CSS preprocessors. Sass stands for "Syntactically Awesome Style Sheets."
- **Features supported by Sass**: Sass supports features like variables, nested CSS rules, modules, mixins, inheritance, and operators for basic mathematical operations

Two Syntaxes Supported by Sass

- **SCSS syntax**: The SCSS (Sassy CSS) expands the basic syntax of CSS. It is the most widely used syntax for Sass. SCSS files have an `.scss` extension.

Here is an example of defining and using a variable in SCSS.

```
$primary-color: #3498eb;

header {
  background-color: $primary-color;
}
```

- **Indented syntax**: The indented syntax was Sass's original syntax and is also known as the "Sass syntax."

Here is an example of defining and using a variable in the indented syntax.

```
$primary-color: #3498eb

header
  background-color: $primary-color
```

Mixins

- **Mixins**: Mixins allow you to group multiple CSS properties and their values under the name and reuse that block of CSS code throughout your stylesheet.

Here is an example of defining a mixin in SCSS syntax. In this case, the mixin is called `center-flex`. It has three CSS properties to center elements using flexbox.

```
@mixin center-flex {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Here is an example of using the mixin you defined.

```
section {
  @include center-flex;
  height: 500px;
  background-color: #3289a8;
}
```

--assignment--

Review the CSS Libraries and Frameworks topics and concepts.

CSS Positioning Review

Working With Floats

- **Definition**: Floats are used to remove an element from its normal flow on the page and position it either on the left or right side of its container. When this happens, the text will wrap around that floated content.

```
float: left;
float: right;
```

- **Clearing Floats**: The `clear` property is used to determine if an element needs to be moved below the floated content. When you have multiple floated elements stacked next to each other, there could be issues with overlap and collapsing in the layouts. So a `clearfix` hack was created to fix this issue.

```
.clearfix::after {
  content: "";
  display: block;
  clear: both;
}
```

Static, Relative and Absolute Positioning

- **Static Positioning**: This is the normal flow for the document. Elements are positioned from top to bottom and left to right one after another.
- **Relative Positioning**: This allows you to use the `top`, `left`, `right` and `bottom` properties to position the element within the normal document flow. You can also use relative positioning to make elements overlap with other elements on the page.

```
.relative {
    position: relative;
    top: 30px;
    left: 30px;
}
```

- **Absolute Positioning**: This allows you to take an element out of the normal document flow, making it behave independently from other elements.

```
.positioned {
  position: absolute;
  top: 30px;
  left: 30px;
  background-color: coral;
}
```

Fixed and Sticky Positioning

- **Fixed Positioning**: When an element is positioned with `position: fixed`, it is removed from the normal document flow and placed relative to the viewport, meaning it stays in the same position even when the user scrolls. This is often used for elements like headers or navigation bars that need to remain visible at all times.

```
.navbar {
  position: fixed;
  top: 0;
  width: 100%;
}
```

- **Sticky Positioning**: This type of positioning will act as a relative positioned element as you scroll down the page. If you specify a `top`, `left`, `right` or `bottom` property, then the element will stop acting like a relatively positioned element and start behaving like a fixed position element.

```
.positioned {
  position: sticky;
  top: 30px;
  left: 30px;
}
```

Working With the `z-index` Property

- **Definition**: The `z-index` property in CSS is used to control the vertical stacking order of positioned elements that overlap on the page.

```
.container {
  position: relative;
  width: 300px;
  height: 300px;
  border: 1px solid black;
}

.box1 {
  position: absolute;
  z-index: 1;
```

```
    background: lightcoral;
    top: 20px;
    left: 20px;
    width: 100px;
    height: 100px;
  }
```

--assignment--

Review the CSS Positioning topics and concepts.

CSS Pseudo-classes Review

User Action Pseudo-classes

- **Pseudo-classes Definition**: These are special CSS keywords that allow you to select an element based on its specific state or position.
- **User Action Pseudo-classes**: These are special keywords that allow you to change the appearance of elements based on user interactions, improving the overall user experience.
- `:active` **Pseudo-class**: This pseudo-class lets you select the active state of an element, like clicking on a button.
- `:hover` **Pseudo-class**: This pseudo-class defines the hover state of an element.
- `:focus` **Pseudo-class**: This pseudo-class applies styles when an element gains focus, typically through keyboard navigation or when a user clicks into a form input.
- `:focus-within` **Pseudo-class**: This pseudo-class is used to apply styles to an element when it or any of its descendants have focus.

Input Pseudo-classes

- **Input Pseudo-classes**: These pseudo-classes are used to target HTML `input` elements based on the state they are in before and after user interaction.
- `:enabled` **Pseudo-class**: This pseudo-class is used to target form buttons or other elements that are currently enabled.
- `:disabled` **Pseudo-class**: This pseudo-class lets you style an interactive element in disabled mode.
- `:checked` **Pseudo-class**: This pseudo-class is used to indicate to the user that it is checked.
- `:valid` **Pseudo-class**: This pseudo-class targets the input fields that meet the validation criteria.
- `:invalid` **Pseudo-class**: This pseudo-class targets the input fields that do not meet the validation criteria.
- `:in-range` **and** `:out-of-range` **Pseudo-classes**: These pseudo-classes apply styles to elements based on whether their values are within or outside specified range constraints.
- `:required` **Pseudo-class**: This pseudo-class targets `input` elements that have the `required` attribute. It signals to the user that they must fill out the field to submit the form.
- `:optional` **Pseudo-class**: This pseudo-class applies styles input elements that are not required and can be left empty.
- `:autofill` **Pseudo-class**: This pseudo-class applies styles to input fields that the browser automatically fills with saved data.

Location Pseudo-classes

- **Location Pseudo-classes**: These pseudo-classes are used for styling links and elements that are targeted within the current document.
- `:any-link` **Pseudo-class**: This pseudo-class is a combination of the `:link` and `:visited` pseudo-classes. So, it matches any anchor element with an href attribute, regardless of whether it's visited or not.

- `:link` **Pseudo-class**: This pseudo-class allows you to target all unvisited links on a webpage. You can use it to style links differently before the user clicks on them.
- `:local-link` **Pseudo-class**: This pseudo-class targets links that point to the same document. It can be useful when you want to differentiate internal links from external ones.
- `:visited` **Pseudo-class**: This pseudo-class targets a link the user has visited.
- `:target` **Pseudo-class**: This pseudo-class is used to apply styles to an element that is the target of a URL fragment.
- `:target-within` **Pseudo-class**: This pseudo-class applies styles to an element when it or one of its descendants is the target of a URL fragment.

Tree-structural Pseudo-classes

- **Tree-structural Pseudo-classes**: These pseudo-classes allow you to target and style elements based on their position within the document tree.
- `:root` **Pseudo-class**: This pseudo-class is usually the root `html` element. It helps you target the highest level in the document so you can apply a common style to the entire document.
- `:empty` **Pseudo-class**: Empty elements, that is, elements with no children other than white space, are also included in the document tree. That's why there's an `:empty` pseudo-class to target empty elements.
- `:nth-child(n)` **Pseudo-class**: This pseudo-class allows you to select elements based on their position within a parent.
- `:nth-last-child(n)` **Pseudo-class**: This pseudo-class enables you to select elements by counting from the end.
- `:first-child` **Pseudo-class**: This pseudo-class selects the first element in a parent element or the document.
- `:last-child` **Pseudo-class**: This pseudo-class selects the last element in a parent element or the document.
- `:only-child` **Pseudo-class**: This pseudo-class selects the only element in a parent element or the document.
- `:first-of-type` **Pseudo-class**: This pseudo-class selects the first occurrence of a specific element type within its parent.
- `:last-of-type` **Pseudo-class**: This pseudo-class selects the last occurrence of a specific element type within its parent.
- `:nth-of-type(n)` **Pseudo-class**: This pseudo-class allows you to select a specific element within its parent based on its position among siblings of the same type.
- `:only-of-type` **Pseudo-class**: This pseudo-class selects an element if it's the only one of its type within its parent.

Functional Pseudo-classes

- **Functional Pseudo-classes**: Functional pseudo-classes allow you to select elements based on more complex conditions or relationships. Unlike regular pseudo-classes which target elements based on a state (for example, `:hover`, `:focus`), functional pseudo-classes accept arguments.
- `:is()` **Pseudo-class**: This pseudo-class takes a list of selectors (ex. `ol`, `ul`) and selects an element that matches one of the selectors in the list.

```
<p class="example">This text will change color.</p>
<p>This text will not change color.</p>
<p>This text will not change color.</p>
<p class="this-works-too">This text will change color.</p>
```

```
p:is(.example, .this-works-too) {
    color: red;
}
```

- `:where()` **Pseudo-class**: This pseudo-class takes a list of selectors (ex. `ol`, `ul`) and selects an element that matches one of the selectors in the list. The difference between `:is` and `:where` is that the latter will have a specificity of 0.

```
:where(h1, h2, h3) {
    margin: 0;
    padding: 0;
}
```

- `:has()` **Pseudo-class**: This pseudo-class is often dubbed the `"parent"` selector because it allows you to style elements that contain child elements specified in the selector list.

```
article:has(h2) {
    border: 2px solid hotpink;
}
```

- `:not()` **Pseudo-class**: This pseudo-class is used to select elements that do not match the provided selector.

```
p:not(.example) {
  color: blue;
}
```

Pseudo-elements

- `::before` **Pseudo-element**: This pseudo-element uses the `content` property to insert cosmetic content like icons just before the element.
- `::after` **Pseudo-element**: This pseudo-element uses the `content` property to insert cosmetic content like icons just after the element.
- `::first-letter` **Pseudo-element**: This pseudo-element targets the first letter of an element's content, allowing you to style it.
- `::marker` **Pseudo-element**: This pseudo-element lets you select the marker (bullet or numbering) of list items for styling.

--assignment--

Review the CSS Pseudo-classes topics and concepts.

CSS Relative and Absolute Units Review

Absolute Units

- `px` **(Pixels)**: This absolute unit is a fixed-size unit of measurement in CSS. It is the most common absolute unit and provides precise control over dimensions. `1px` is always equal to 1/96th of an inch.
- `in` **(Inch)**: This absolute unit is equal to 96px.
- `cm` **(Centimeters)**: This absolute unit is equal to 25.2/64 of an inch.
- `mm` **(Millimeters)**: This absolute unit is equal to 1/10th of a centimeter.
- `q` **(Quarter-Millimeters)**: This absolute unit is equal to 1/40th of a centimeter.
- `pc` **(Picas)**: This absolute unit is equal to 1/6th of an inch.
- `pt` **(Points)**: This absolute unit is equal to 1/72th of an inch.

Relative Units

- **Percentages**: These relative units allow you to define sizes, dimensions, and other properties as a proportion of their parent element. For example, if you set `width: 50%;` on an element, it will occupy half the width of its parent container.
- **`em` Unit**: These units are relative to the font size of the element. If you are using `ems` for the `font-size` property, the size of the text will be relative to the font size of the parent element.
- **`rem` Unit**: These units are relative to the font size of the root element, which is the `html` element.
- **`vh` Unit**: `vh` stands for `"viewport height"` and `1vh` is equal to 1% of the viewport's height.
- **`vw` Unit**: `vw` stands for `"viewport width"` and `1vw` is equal to 1% of the viewport's width.

`calc` Function

- **`calc()` Function**: With the `calc()` function, you can perform calculations directly within your stylesheets to determine property values dynamically. This means that you can create flexible and responsive user interfaces by calculating dimensions based on the viewport size or other elements.

--assignment--

Review the CSS Relative and Absolute Units topics and concepts.

CSS Typography Review

Introduction to Typography

- **Definition**: Typography is the art of choosing the right fonts and format to make text visually appealing and easy to read. "Type" refers to how the individual characters are designed and arranged.
- **Typeface Definition**: A typeface is the overall design and style of a set of characters, numbers, and symbols. It's like a blueprint for a family of fonts.
- **Font Definition**: A font is a specific variation of a typeface with specific characteristics, such as size, weight, style, and width.
- **Serif Typeface**: This typeface has a classical style with small lines at the end of characters. Serif typefaces are commonly used for printed materials, like books.
- **Sans Serif Typeface**: This typeface has a more modern look, without the small lines at the end of characters. Sans Serif typefaces are commonly used in digital design because they are easy to read on screen. Some examples include Helvetica, Arial, and Roboto.
- **Font Weight**: This is the thickness of the characters, including light, regular, bold, and black.
- **Font Style**: This is the slant and orientation of the characters, like italic and oblique.
- **Font Width**: This is the horizontal space occupied by characters, like condensed and expanded.
- **Baseline**: This is the imaginary line on which most characters rest.
- **Cap Height**: This is the height of uppercase letters, measured from the baseline to the top.
- **X-height**: This is the average height of lowercase letters, excluding ascenders and descenders.
- **Ascenders**: These are the parts of lowercase letters that extend above the x-height, such as the tops of the letters 'h', 'b', 'd', and 'f'.
- **Descenders**: These are the parts of lowercase letters that extend below the baseline, such as the tails of 'y', 'g', 'p', and 'q'.
- **Kerning**: This is how space is adjusted between specific pairs of characters to improve their readability and aesthetics. For example, reducing the space between the letters A and V.
- **Tracking**: This is how space is adjusted between all characters in a block of text. It's essentially the distance between the characters. It affects how dense and open the text will be.
- **Leading**: This is the vertical space between lines of text. It's measured from the baseline of one line to the baseline of the next line.
- **Best Practices with Typography**: You should choose clear and simple fonts to make your designs easy to understand. This is particularly important for the main text of your website. Users are more likely to engage with your content if the font is easy to read. You should use two or three fonts at most to create a visual consistency. Using too many fonts can make the text more difficult to read

and weaken your brand's identity. This can also impact the user experience by increasing the load time of the website. You can use font size to create a visual hierarchy for headings, subheadings, paragraphs, and other elements. For example, the main heading on a webpage should have a larger font, followed by subheadings with smaller font sizes. This will give every element in the hierarchy a specific font size that helps users navigate through the structure visually.

CSS `font-family` Property

- **Definition**: A font family is a group of fonts that share a common design. All the fonts that belong to the same family are based on the same core typeface but they also have variations in their style, weight, and width. You can specify multiple font families in order of priority, from highest to lowest, by separating them with commas. These alternative fonts will act as fallback options. You should always include a generic font family at the end of the font-family list.

```
font-family: Arial, Lato;
```

- **Common Font Families**: Here are some common font families used in CSS: serif, sans-serif, monospace, cursive, fantasy

Web Safe Fonts

- **Definition**: These fonts are a subset of fonts that are very likely to be installed on a computer or device. When the browser has to render a font, it tries to find the font file on the user's system. But if the font is not found, it will usually fall back to a default system font.
- **Web Safe Fonts**:
  - Sans-serif fonts are commonly used for web development because they don't have small "feet" or lines at the end of the characters, so they're easy to read on screen. Some examples of web-safe sans-serif fonts are: Arial, Verdana, and Trebuchet MS.
  - In contrast, serif fonts do have small "feet" at the end of the characters, so they're commonly used for traditional print. Web safe serif fonts include: Times New Roman and Georgia.

At-rules and the `@font-face` At-rule

- **Definition**: At-rules are statements that provide instructions to the browser. You can use them to define various aspects of the stylesheet, such as media queries, keyframes, font faces, and more.
- **`@font-face`**: This allows you to define a custom font by specifying the font file, format, and other important properties, like weight and style. For the `@font-face` at-rule to be valid, you also need to specify the `src` property. This property contains references to the font resources.

```
@font-face {
  font-family: "MyCustomFont";
  src: url("path/to/font.woff2"),
       url("path/to/font.woff"),
       url("path/to/font.otf");
}
```

- **Font Formats**: For each font resource, you can also specify the format. This is optional. It's a hint for the browser on the font format. If the format is omitted, the resource will be downloaded and the format will be detected after it's downloaded. If the format is invalid, the resource will not be downloaded. Possible font formats include `collection`, `embedded-opentype`, `opentype`, `svg`, `truetype`, `woff`, and `woff2`.

```
@font-face {
  font-family: "MyCustomFont";
```

```
  src: url("path/to/font.woff2") format("woff2"),
       url("path/to/font.otf") format("opentype"),
       url("path/to/font.woff") format("woff");
}
```

- **`woff` and `woff2`**: `woff` stands for "Web Open Font Format." It's a widely used font format developed by Mozilla in collaboration with Type Supply, LettError, and other organizations. The difference between `woff` and `woff2` is the algorithm used to compress the data.
- **OpenType**: This is a format for scalable computer fonts developed by Microsoft and Adobe that allows users to access additional features in a font. It's widely used across major operating systems.
- **`tech()`**: This is used to specify the technology of the font resource. This is optional.

```
@font-face {
  font-family: "MyCustomFont";
  src: url("path/to/font.woff2") format("woff2"),
       url("path/to/font.otf") format("opentype") tech(color-COLRv1),
       url("path/to/font.woff") format("woff");
}
```

Working With External Fonts

- **Definition**: An external font is a font file that is not included directly within your project files.They're usually hosted on a separate server. To include these external fonts inside your project, you can use a `link` element or `@import` statement inside your CSS.
- **Google Fonts**: This is a Google service that offers a collection of fonts, many of which are designed specifically for web development.

Here is an example using the `link` element:

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?
family=Roboto:ital,wght@0,100;0,300;0,400;0,500;0,700;0,900;1,100;1,300;1,400
;1,500;1,700;1,900&display=swap" rel="stylesheet">
```

```
.roboto-thin {
  font-family: "Roboto", sans-serif;
  font-weight: 100;
  font-style: normal;
}
```

Here is an example using the `@import` statement:

```
@import url('https://fonts.googleapis.com/css2?
family=Roboto:ital,wght@0,100;0,300;0,400;0,500;0,700;0,900;1,100;1,300;1,400
;1,500;1,700;1,900&display=swap');
```

- **Font Squirrel**: This is another popular resource that you can use for adding custom external fonts to your projects.

`text-shadow` Property

- **Definition**: This property is used to apply shadows to text. You need to specify the X and Y offset, which represent the horizontal and vertical distance of the shadow from the text, respectively. These values are required. Positive values of the X offset and Y offset will move the shadow right and down, respectively, while negative values will move the shadow left and up.

```
p {
  text-shadow: 3px 2px;
}
```

- **Shadow Color**: You can also customize the color of the shadow by specifying this value before or after the offset. If the color is not specified, the browser will determine the color automatically, so it's usually best to set it explicitly.

```
p {
  text-shadow: 3px 2px #00ffc3;
}
```

- **Blur Radius**: The blur radius is optional but will make the shadow look a lot smoother and more subtle. The default value of the radius blur is zero. The higher the value, the bigger the blur, which means that the shadow becomes lighter.

```
p {
  text-shadow: 3px 2px 3px #00ffc3;
}
```

- **Applying Multiple Text Shadows**: The text can have more than one shadow. The shadows will be applied in layers, from front to back, with the first shadow at the top.

```
p {
  text-shadow:
    3px 2px 3px #00ffc3,
    -3px -2px 3px #0077ff,
    5px 4px 3px #dee7e5;
}
```

--assignment--

Review the CSS Typography topics and concepts.

CSS Variables Review

CSS Custom Properties (CSS Variables)

- **Definition**: CSS custom properties, also known as CSS variables, are entities defined by CSS authors that contain specific values to be reused throughout a document. They are a powerful feature that allows for more efficient, maintainable, and flexible stylesheets. Custom properties are particularly useful in creating themeable designs. You can define a set of properties for different themes:

```
:root {
  --bg-color: white;
```

```
  --text-color: black;
}

.dark-theme {
  --bg-color: #333;
  --text-color: white;
}

body {
  background-color: var(--bg-color);
  color: var(--text-color);
}
```

The `@property` Rule

- **Definition**: The `@property` rule is a powerful CSS feature that allows developers to define custom properties with greater control over their behavior, including how they animate and their initial values.

```
@property --property-name {
  syntax: '<type>';
  inherits: true | false;
  initial-value: <value>;
}
```

- `--property-name`: This is the name of the custom property you're defining. Like all custom properties, it must start with two dashes. `--property-name` can be things like `<color>`, `<length>`, `<number>`, `<percentage>`, or more complex types.
- `syntax`: This defines the type of the property.
- `inherits`: This specifies whether the property should inherit its value from its parent element.
- `initial-value`: This sets the default value of the property.
- **Gradient Example Using the `@property` Rule**: This example creates a gradient that smoothly animates when the element is hovered over.

```
<div class="gradient-box"></div>
```

```
@property --gradient-angle {
  syntax: "<angle>";
  inherits: false;
  initial-value: 0deg;
}

.gradient-box {
  width: 100px;
  height: 100px;
  background: linear-gradient(var(--gradient-angle), red, blue);
  transition: --gradient-angle 0.5s;
}

.gradient-box:hover {
  --gradient-angle: 90deg;
}
```

- **Fallbacks**: When using the custom property, you can provide a fallback value using the `var()` function, just as you would with standard custom properties:

```
.button {
  background-color: var(--main-color, #3498db);
}
```

--assignment--

Review the CSS Variables topics and concepts.

Data Structures Review

Algorithms and Big O Notation

- **Algorithms**: A set of unambiguous instructions for solving a problem or carrying out a task. Algorithms must finish in a finite number of steps and each step must be precise and unambiguous.
- **Big O Notation**: Describes the worst-case performance, or growth rate, of an algorithm as the input size increases. It focuses on how resource usage grows with input size, ignoring constant factors and lower-order terms.

Common Time Complexities

- **O(1) - Constant Time**: Algorithm takes the same amount of time regardless of input size.

```
def check_even_or_odd(number):
    if number % 2 == 0:
        return 'Even'
    else:
        return 'Odd'
```

- **O(log n) - Logarithmic Time**: Time increases slowly as input grows. Common in algorithms that repeatedly reduce problem size by a fraction (like Binary Search).
- **O(n) - Linear Time**: Running time increases proportionally to input size.

```
for grade in grades:
    print(grade)
```

- **O(n log n) - Log-Linear Time**: Common time complexity of efficient sorting algorithms like Merge Sort and Quick Sort.
- **O(n^2) - Quadratic Time**: Running time increases quadratically. Often seen in nested loops.

```
for i in range(n):
    for j in range(n):
        print("Hello, World!")
```

Space Complexity

- **O(1) - Constant Space**: Algorithm uses same amount of memory regardless of input size.
- **O(n) - Linear Space**: Memory usage grows proportionally with input size.
- **O(n^2) - Quadratic Space**: Memory usage grows quadratically with input size.

Problem-Solving Techniques

- **Understanding the Problem**: Read the problem statement multiple times. Identify the input, expected output, and how to transform input to output.
- **Pseudocode**: High-level description of algorithm logic that is language-independent. Uses common written language mixed with programming constructs like `IF`, `ELSE`, `FOR`, `WHILE`.

```
GET original_string
SET reversed_string = ""
FOR EACH character IN original_string:
  ADD character TO THE BEGINNING OF reversed_string
DISPLAY reversed_string
```

- **Edge Cases**: Specific, valid inputs that occur at the boundaries of what an algorithm should handle. Always consider and test edge cases.

Arrays

- **Static Arrays**: Have a fixed size determined at initialization. Elements stored in adjacent memory locations. Size cannot be changed during program execution.
- **Dynamic Arrays**: Can grow or shrink automatically during program execution. Handle resizing through automatic copying to larger arrays when needed.

Python Lists (Dynamic Arrays)

```
numbers = [3, 4, 5, 6]

# Access elements
numbers[0]  # 3

# Update elements
numbers[2] = 16

# Add elements
numbers.append(7)
numbers.insert(3, 15)  # Insert at specific index

# Remove elements
numbers.pop(2)  # Remove at specific index
numbers.pop()   # Remove last element
```

Time Complexities for Dynamic Arrays

- **Access**: O(1)
- **Insert at end**: O(1) average, O(n) when resizing needed
- **Insert in middle**: O(n)
- **Delete**: O(n) for middle, O(1) for end

Stacks

- **Stacks**: Last-In, First-Out (LIFO) data structure. Elements added and removed from the top only.
- **Push Operation**: Adding an element to the top of the stack. Time complexity: O(1).
- **Pop Operation**: Removing an element from the top of the stack. Time complexity: O(1).

```

```

```
# Using Python list as stack
stack = []

# Push operations
stack.append(1)
stack.append(2)
stack.append(3)

# Pop operations
top_element = stack.pop()  # Returns 3
```

Queues

- **Queues**: First-In, First-Out (FIFO) data structure. Elements added to the back and removed from the front.
- **Enqueue Operation**: Adding an element to the back of the queue. Time complexity: O(1).
- **Dequeue Operation**: Removing an element from the front of the queue. Time complexity: O(1).

```
from collections import deque

# Using deque for efficient queue operations
queue = deque()

# Enqueue operations
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeue operations
first_element = queue.popleft()  # Returns 1
```

Linked Lists

- **Linked Lists**: Linear data structure where each node contains data and a reference to the next node. Nodes are connected like a chain.

Singly Linked Lists

- **Structure**: Each node has data and one reference to the next node.
- **Traversal**: Can only move forward from head to tail.
- **Head Node**: First node in the list, usually the only directly accessible node.
- **Tail Node**: Last node in the list, points to `None`.

Operations and Time Complexities

- **Insert at beginning**: O(1)
- **Insert at end**: O(n) - must traverse to end
- **Insert in middle**: O(n) - must traverse to position
- **Delete from beginning**: O(1)
- **Delete from end**: O(n) - must traverse to find previous node
- **Delete from middle**: O(n) - must traverse to find node

Doubly Linked Lists

- **Structure**: Each node has data and two references: next node and previous node.
- **Traversal**: Can move in both directions.
- **Memory**: Requires more memory than singly linked lists due to extra reference.

Hash Maps and Sets

Maps and Hash Maps

- **Map (Abstract Data Type)**: Manages collections of key-value pairs. Every key must be unique, but values can be repeated.
- **Hash Map**: Concrete implementation of map ADT using hashing technique. Uses hash function to generate hash values for keys, which determine storage location in underlying array.

Python Dictionaries (Hash Maps)

```
# Creating dictionaries
my_dictionary = {
    "A": 1,
    "B": 2,
    "C": 3
}

# Alternative creation
my_dictionary = dict(A=1, B=2, C=3)

# Access and modify
value = my_dictionary["A"]  # 1
my_dictionary["A"] = 4      # Update value
del my_dictionary["A"]      # Remove key-value pair

# Check membership
"C" in my_dictionary

# Get keys, values, items
my_dictionary.keys()
my_dictionary.values()
my_dictionary.items()
```

Time Complexities for Hash Maps

- **Average case**: O(1) for insert, get, delete
- **Worst case**: O(n) when many hash collisions occur

Sets

- **Sets**: Unordered collections of unique elements. No duplicates allowed, no specific order maintained.
- **Immutable Elements Only**: Sets can only contain immutable data types (numbers, strings, tuples) because hash values must remain constant.

```
# Creating sets
numbers = {1, 2, 3, 4}
empty_set = set()  # Must use set(), not {}

# Add and remove elements
numbers.add(5)
numbers.remove(4)      # Raises KeyError if not found
numbers.discard(4)     # No error if not found

# Set operations
set_a = {1, 2, 3, 4}
set_b = {2, 3, 4, 5, 6}
```

```
# Union, intersection, difference, symmetric difference
set_a.union(set_b)                    # or set_a | set_b
set_a.intersection(set_b)             # or set_a & set_b
set_a.difference(set_b)               # or set_a - set_b
set_a.symmetric_difference(set_b)     # or set_a ^ set_b

# Subset and superset checks
set_a.issubset(set_b)
set_a.issuperset(set_b)
set_a.isdisjoint(set_b)

# Membership testing
5 in numbers
```

Time Complexities for Sets

- **Average case**: O(1) for add, remove, membership testing
- **Worst case**: O(n) due to hash collisions

Hash Collisions

- **Hash Collision**: Occurs when two different keys produce the same hash value.
- **Collision Resolution Strategies**:
  - **Chaining**: Each array index points to a linked list storing all elements with same hash value
  - **Open Addressing**: Search for next available index using predefined sequence

When to Use Each Data Structure

- **Lists**: When you need ordered, indexed access and don't know size in advance
- **Stacks**: For LIFO operations (undo functionality, expression evaluation, backtracking)
- **Queues**: For FIFO operations (task scheduling, breadth-first search)
- **Linked Lists**: When frequent insertion/deletion at beginning, unknown size, no random access needed
- **Hash Maps**: For fast key-value lookups, counting occurrences, caching
- **Sets**: For uniqueness checking, mathematical set operations, removing duplicates

--assignment--

Review the Data Structures topics and concepts.

Debugging JavaScript Review

Common Types of Error Messages

- **SyntaxError**: These errors happen when you write something incorrectly in your code, like missing a parenthesis, or a bracket. Think of it like a grammar mistake in a sentence.

```
const arr = ["Beau", "Quincy" "Tom"]
```

- **ReferenceError**: There are several types of Reference Errors, triggered in different ways. The first type of reference error would be not defined variables. Another example of a ReferenceError is trying to access a variable, declared with `let` or `const`, before it has been defined.

```
```

```
console.log(num);
const num = 50;
```

- **TypeError**: These errors occur when you try to perform an operation on the wrong type.

```
const developerObj = {
  name: "Jessica",
  country: "USA",
  isEmployed: true
};

developerObj.map()
```

- **RangeError**: These errors happen when your code tries to use a value that's outside the range of what JavaScript can handle.

```
const arr = [];
arr.length = -1;
```

The `throw` Statement

- **Definition**: The `throw` statement in JavaScript is used to throw a user defined exception. An exception in programming, is when an unexpected event happens and disrupts the normal flow of the program.

```
function validateNumber(input) {
  if (typeof input !== "number") {
    throw new TypeError("Expected a number, but received " + typeof input);
  }
  return input * 2;
}
```

`try...catch...finally`

- **Definition**: The `try` block is used to wrap code that might throw an error. It acts as a safe space to try something that could fail. The `catch` block captures and handles errors that occur in the try block. You can use the error object inside catch to inspect what went wrong. The `finally` block runs after the try and catch blocks, regardless of whether an error occurred. It's commonly used for cleanup tasks, such as closing files or releasing resources.

```
function processInput(input) {
  if (typeof input !== "string") {
    throw new TypeError("Input must be a string.");
  }

  return input.toUpperCase();
}

try {
  console.log("Starting to process input...");
  const result = processInput(9);
  console.log("Processed result:", result);
} catch (error) {
```

```
  console.error("Error occurred:", error.message);
}
```

Debugging Techniques

- `debugger` **Statement**: This statement lets you pause your code at a specific line to investigate what's going on in the program.

```
let firstNumber = 5;
let secondNumber = 10;

debugger; // Code execution pauses here
let sum = firstNumber + secondNumber;

console.log(sum);
```

- **Breakpoints**: Breakpoints let you pause the execution of your code at a specific line of your choice. After the pause, you can inspect variables, evaluate expressions, and examine the call stack.
- **Watchers**: Watch expressions lets you monitor the values of variables or expressions as the code runs even if they are out of the current scope.
- **Profiling**: Profiling helps you identify performance bottlenecks by letting you capture screenshots and record CPU usage, function calls, and execution time.
- `console.dir()`: This method is used to display an interactive list of the properties of a specified JavaScript object. It outputs a hierarchical listing that can be expanded to see all nested properties.

```
console.dir(document);
```

- `console.table()`: This method displays tabular data as a table in the console. It takes one mandatory argument, which must be an array or an object, and one optional argument to specify which properties (columns) to display.

--assignment--

Review the Debugging JavaScript topics and concepts.

Design Fundamentals Review

Design Terminology

- **Layout**: This is how visual elements are arranged on a page or screen to communicate a message. These elements may include text, images, and white space.
- **Alignment**: This is how the elements are placed in relation to one another. Using alignment correctly is helpful for making the design look clean and organized.
- **Composition**: This is the act of arranging elements to create a harmonious design. It determines how elements like images, text, and shapes relate to each other and contribute to the design in an artistic way.
- **Balance**: This is how visual weight is distributed within a composition. Designers aim to create an equilibrium through symmetrical or asymmetrical arrangements.
- **Scale**: This refers to comparing the dimensions or size of one element to that of another.
- **Hierarchy**: This establishes the order of importance of the elements in a design. It's about making sure that the most important information is noticed first.

- **Contrast**: This is the process of creating clear distinctions between the elements. You can do this through variations in color, size, shape, texture, or any other visual characteristic. Strong contrast is also helpful for improving readability.
- **White Space(negative space)**: This is the empty space in a design. It's the area surrounding the elements.
- **UI(User Interface)**: UI includes the visual and interactive elements that users can see on their screens, like icons, images, text, menus, links, and buttons.
- **UX(User Experience)**: UX is about how users feel when using a product or service. An application with a well-design user experience is intuitive, easy to use, efficient, accessible, and enjoyable.
- **Design Brief**: This is a document that outlines the objectives, goals, and requirements of a project. It is a roadmap that guides the design process and ensures that the final product meets the needs of the client.
- **Vector Based Design**: This involves creating digital art using mathematical formulas to define lines, shapes, and colors.
- **Prototyping**: This refers to the process of creating an interactive model of a product or user interface.

UI Design Fundamentals

- **Good Contrast for Background and Foreground Colors**: It is important to ensure that the background and foreground colors have good contrast to make the text readable. The Web Content Accessibility Guidelines (WCAG) recommend a minimum contrast ratio of 4.5:1 for normal text and 3:1 for large text.
- **Good Visual Hierarchy in Design**: A strong visual hierarchy can provide a clear path for the eye to follow, ensuring that the information you convey is consumed in the order that you intend.
- **Responsive Images**: Responsive images are images that scale to fit the size of the screen they are being viewed on. This is important because it ensures that your images look good on all devices, from desktops to mobile phones.
- **Progressive Enhancement**: This is a design approach that ensures all users, regardless of browser or device, can access the essential content and functionality of an application.
- **User-centered Design**: This is an approach that prioritizes the end user, from their needs to their preferences and limitations. The goal of user-centered design is to craft a web page that is intuitive, efficient to use, and pleasing for your users to interact with.
- **User Research**: This is the systematic study of the people who use your product. The goal is to measure user needs, behaviors, and pain points.
- **Exit Interviews**: This is a survey you can give to users when they cancel their accounts. It can help you understand why users are leaving and what you can do to reduce churn.
- **User Testing**: This refers to the practice of capturing data from users as they interface with your application.
- **A/B Testing**: This is the process of shipping a new feature to a randomly selected subset of your user base. You can then leverage analytics data to determine if the feature is beneficial.
- **User Requirements**: This refers to the stories or rubric that your application needs to follow. User requirements might be defined by user research or industry standards. They can even be defined by stakeholder input.
- **Progressive Disclosure**: This is a design pattern used to only show users relevant content based on their current activity and hide the rest. This is done to reduce cognitive load and make the user experience more intuitive.
- **Deferred/Lazy Registration**: This is a UI design pattern that allows users to browse and interact with your application without having to register.

Design Best Practices

- **Dark Mode**: This is a special feature on web applications where you can change the default light color scheme to a dark color scheme. You should use desaturated colors in dark mode. Desaturated colors are colors that are less intense and have a lower saturation level.
- **Breadcrumbs**: This is a navigation aid that shows the user where they are in the site's hierarchy. It is best to place breadcrumbs at the top of the page so users can easily find it. Also, you want to make

sure the breadcrumbs are large enough to be easily read, but not so large that they take up too much space on the page.

- **Card Component**: Your card component should be simple in design, not visually cluttered or display too much information. For media, make sure to choose high-quality images and videos to enhance the user experience.
- **Infinite Scroll**: This is a design pattern that loads more content as the user scrolls down the page. You should consider using a load more button because it gives a user control over when they want to see more content. You can also add a back button so users have the ability to go back to the previous page without having to scroll all the way back up.
- **Modal Dialog**: This is a type of pop-up that will display on top of existing page content. Usually the background content will have a dim color overlay in order to help the user better focus on the modal content. Also, it is always a good idea to allow the user to click outside of the modal to close it. When you use the HTML `dialog` element, you will get a lot of the functionality and accessibility benefits built in.
- **Progress Indication for Form Registration**: This is a way to show users how far they are in a process. It can be used in forms, registration, and setup processes. Your design should be simple, easy to find, and make it possible to go back to previous steps.
- **Shopping Cart**: Carts are a place for user to see what item they have already selected on an e-commerce platform. Your carts should always be visible to the user, use a common icon like a cart, bag or basket, and have a clear call-to-action button for users to proceed to checkout.

Common Design Tools

- **Figma**: This cloud-based tool specializes in User Interface and User Experience (UI / UX) design. It enables design and development teams to collaborate from anywhere, offering built-in features including Vector-based design, automatic layout, a commenting and feedback system and more.
- **Sketch**: This is a popular design tool used for its intuitive interface and simplicity, making it ideal for developers who want to quickly create prototypes. It's also widely used by designers for tasks like creating UIs, icons, and web layouts.
- **Adobe XD**: This is a vector-based design and prototyping tool for UI/UX design, known for its seamless integration with other Adobe apps like Photoshop, Illustrator, and After Effects.
- **Canva**: This tool allows you to create a wide range of visual content, including posters, cover photos, presentations, short videos, and more. Its user-friendly and simple design makes it ideal for beginners.

--assignment--

Review the Design Fundamentals topics and concepts.

Dictionaries and Sets Review

Dictionaries

- **Dictionaries**: Dictionaries are built-in data structures that store collections of key-value pairs. Keys need to be immutable data types. This is the general syntax of a Python dictionary:

```
dictionary = {
    key1: value1,
    key2: value2
}
```

- **`dict()` Constructor**: The `dict()` constructor is an alternative way to build the dictionary. You pass a list of tuples as an argument to the `dict()` constructor. These tuples contain the key as the first element and the value as the second element.

```
pizza = dict([('name', 'Margherita Pizza'), ('price', 8.9),
('calories_per_slice', 250), ('toppings', ['mozzarella', 'basil'])])
```

- **Bracket Notation**: To access the value of a key-value pair, you can use the syntax known as bracket notation.

```
dictionary[key]
```

Common Dictionary Methods

- **`get()` Method**: The `get()` method retrieves the value associated with a key. It's similar to the bracket notation, but it lets you set a default value, preventing errors if the key doesn't exist.

```
dictionary.get(key, default)
```

- **`keys()` and `values()` Methods**: The `keys()` and `values()` methods return a view object with all the keys and values in the dictionary, respectively. A view object is a way to see the content of a dictionary without creating a separate copy of the data.

```
pizza = {
    'name': 'Margherita Pizza',
    'price': 8.9,
    'calories_per_slice': 250
}

pizza.keys()
# dict_keys(['name', 'price', 'calories_per_slice'])

pizza.values()
# dict_values(['Margherita Pizza', 8.9, 250])
```

- **`items()` Method**: The `items()` method returns a view object with all the key-value pairs in the dictionary, including both the keys and the values.

```
pizza.items()
# dict_items([('name', 'Margherita Pizza'), ('price', 8.9),
('calories_per_slice', 250)])
```

- **`clear()` Method**: The `clear()` method removes all the key-value pairs from the dictionary.

```
pizza.clear()
```

- **`pop()` Method**: The `pop()` method removes the key-value pair with the key specified as the first argument and returns its value. If the key doesn't exist, it returns the default value specified as the second argument. If the key doesn't exist and the default value is not specified, a `KeyError` is raised.

```
pizza.pop('price', 10)
pizza.pop('total_price') # KeyError
```

- **`popitem()` Method**: In Python 3.7 and above, the `popitem()` method removes the last inserted item.

```
pizza.popitem()
```

- **`update()` Method**: The `update()` method updates the key-value pairs with the key-value pairs of another dictionary. If they have keys in common, their values are overwritten. New keys will be added to the dictionary as new key-value pairs.

```
pizza.update({ 'price': 15, 'total_time': 25 })
```

Looping Over a Dictionary

- **Iterating Over Values**: If you need to iterate over the values in a dictionary, you can write a `for` loop with `values()` to get all the values of a dictionary.

```
products = {
    'Laptop': 990,
    'Smartphone': 600,
    'Tablet': 250,
    'Headphones': 70,
}

for price in products.values():
    print(price)
```

Output:

```
990
600
250
70
```

- **Iterating Over Keys**: If you need to iterate over the keys in the `products` dictionary above, you can write `products.keys()` or `products` directly.

```
for product in products.keys():
    print(product)

for product in products:
    print(product)
```

Output:

```
Laptop
Smartphone
Tablet
Headphones
```

- **Iterating Over Key-Value Pairs**: If you need to iterate over the keys and their corresponding values simultaneously, you can iterate over `products.items()`. You get individual tuples with the keys and their corresponding values.

```
for product in products.items():
    print(product)
```

Output:

```
('Laptop', 990)
('Smartphone', 600)
('Tablet', 250)
('Headphones', 70)
```

To store the key and value in separate loop variables, you need to separate them with a comma. The first variable stores the key, and the second stores the value.

```
for product, price in products.items():
    print(product, price)
```

Output:

```
Laptop 990
Smartphone 600
Tablet 250
Headphones 70
```

- **`enumerate()` Function**: If you need to iterate over a dictionary while keeping track of a counter, you can call the `enumerate()` function. The function returns an `enumerate` object, which assigns an integer to each item, like a counter. You can start the counter from any number, but by default, it starts from 0.

Assigning the index and item to separate loop variables is the common way to use `enumerate()`. For example, with `products.items()`, you can get the entire key-value pair in addition to the index:

```
for index, product in enumerate(products.items()):
    print(index, product)
```

Output:

```
0 ('Laptop', 990)
1 ('Smartphone', 600)
2 ('Tablet', 250)
3 ('Headphones', 70)
```

To customize the initial value of the count, you can pass a second argument to `enumerate()`. For example, here we are starting the count from 1.

```
for index, product in enumerate(products.items(), 1):
    print(index, product)
```

Output:

```
1 ('Laptop', 990)
2 ('Smartphone', 600)
3 ('Tablet', 250)
4 ('Headphones', 70)
```

Sets

- **Sets**: Sets are built-in data structures in Python that do not allow duplicate values. Sets are mutable and unordered, which means that their elements are not stored in any specific order, so you cannot use indices or keys to access them. Also, sets can only contain values of immutable data types, like numbers, strings, and tuples.
- **Defining a Set**: To define a set, you need to write its elements within curly brackets and separate them with commas.

```
my_set = {1, 2, 3, 4, 5}
```

- **Defining an Empty Set**: If you need to define an empty set, you must use the `set()` function. Only writing empty curly braces will automatically create a dictionary.

```
set() # Set
{}    # Dictionary
```

Common Set Methods

- **`add()` Method**: You can add an element to a set with the `add()` method, passing the new element as an argument.

```
my_set.add(6)
```

- **`remove()` and `discard()` Methods**: To remove an element from a set, you can either use the `remove()` method or the `discard()` method, passing the element you want to remove as an argument. The `remove()` method will raise a `KeyError` if the element is not found while the `discard()` method will not.

```
my_set.remove(4)
my_set.discard(4)
```

- **`clear()` method**: The `clear()` method removes all the elements from the set.

```
my_set.clear()
```

Mathematical Set Operations

- **`issubset()` and `issuperset()` Methods**: The `issubset()` and the `issuperset()` methods check if a set is a subset or superset of another set, respectively.

```
my_set = {1, 2, 3, 4, 5}
your_set = {2, 3, 4, 5}

print(your_set.issubset(my_set)) # True
print(my_set.issuperset(your_set)) # True
```

- **`isdisjoint()` Method**: The `isdisjoint()` method checks if two sets are disjoint, if they don't have elements in common.

```
print(my_set.isdisjoint(your_set)) # False
```

- **Union Operator (`|`)**: The union operator `|` returns a new set with all the elements from both sets.

```
my_set | your_set # {1, 2, 3, 4, 5, 6}
```

- **Intersection Operator (`&`)**: The intersection operator `&` returns a new set with only the elements that the sets have in common.

```
my_set & your_set # {2, 3, 4}
```

- **Difference Operator (`-`)**: The difference operator `-` returns a new set with the elements of the first set that are not in the other sets.

```
my_set - your_set # {1, 5}
```

- **Symmetric Difference Operator (`^`)**: The symmetric difference operator `^` returns a new set with the elements that are either on the first or the second set, but not both.

```
my_set ^ your_set # {1, 5, 6}
```

- **`in` Operator**: You can check if an element is in a set or not with the `in` operator.

```
print(5 in my_set)
```

Python Standard Library

- **Python Standard Library**: A library gives you pre-written and reusable code, like functions, classes, and data structures, that you can reuse in your projects. Python has an extensive standard library with built-in modules that implement standardized solutions for many problems and tasks. Some examples of popular built-in modules are `math`, `random`, `re` (short for "regular expressions"), and `datetime`.

Import Statement

- **Import Statement**: To access the elements defined in built-in modules, you use an import statement. Import statements are generally written at the top of the file. Import statements work the same for functions, classes, constants, variables, and any other elements defined in the module.

- **Basic Import Statement**: You can use the `import` keyword followed by the name of the module:

```
import module_name
```

Then, if you need to call a method from that module, you would use dot notation, with the name of the module followed by the name of the method.

```
module_name.method_name()
```

For example, you would write the following in your code to import the `math` module and get the square root of 36:

```
import math

math.sqrt(36)
```

- **Importing a Module with a Different Name**: If you need to import the module with a different name (also known as an "alias"), you can use `as` followed by the alias at the end of the import statement. This is often used for long module names or to avoid naming conflicts.

```
import module_name as module_alias
```

For example, to refer to the `math` module as `m` in your code, you can assign an alias like this:

```
import math as m
```

Then, you can access the elements of the module using the alias:

```
m.sqrt(36)
```

- **Importing Specific Elements**: If you don't need everything from a module, you can import specific elements using `from`. In this case, the import statement starts with `from`, followed by the module name, then the `import` keyword, and finally the names of the elements you want to import.

```
from module_name import name1, name2
```

Then, you can use these names without the module prefix in your Python script. For example:

```
from math import radians, sin, cos

angle_degrees = 40
angle_radians = radians(angle_degrees)

sine_value = sin(angle_radians)
cos_value = cos(angle_radians)

print(sine_value) # 0.6427876096865393
print(cos_value)  # 0.766044443118978
```

This is helpful, but it can result in naming conflicts if you already have functions or variables with the same name. Keep it in mind when choosing which type of import statement you want to use.

If you need to assign aliases to these names, you can do so as well, using the `as` keyword followed by the alias.

```
from module_name import name1 as alias1, name2 as alias2
```

- **Import Statement with Asterisk (`*`)**: The asterisk tells Python that you want to import everything in that module, but you want to import it so that you don't need to use the name of the module as a prefix.

```
from module_name import *
```

For example, if you this to import the `math` module, you'll be able to call any function defined in that module without specifying the name of the module as a prefix.

```
from math import *
print(sqrt(36))  # 6.0
```

However, this is generally discouraged because it can lead to namespace collisions and make it harder to know where names come from.

`if __name__ == '__main__'`

- **`__name__` Variable**: `__name__` is a special built-in variable in Python. When a Python file is executed directly, Python sets the value of this variable to the string `"__main__"`. But if the Python file is imported as a module into another Python script, the value of the `__name__` variable is set to the name of that module.

This is why you'll often find this conditional in Python scripts. It contains the code that you only want to run **only** if the Python script is running as the main program.

```
if __name__ == '__main__':
    # Code
```

--assignment--

Review the Dictionaries and Sets topics and concepts.

DOM Manipulation and Click Events with JavaScript Review

Working with the DOM and Web APIs

- **API**: An API (Application Programming Interface) is a set of rules and protocols that allow software applications to communicate with each other and exchange data efficiently.

- **Web API**: Web APIs are specifically designed for web applications. These types of APIs are often divided into two main categories: browser APIs and third-party APIs.
- **Browser APIs**: These APIs expose data from the browser. As a web developer, you can access and manipulate this data using JavaScript.
- **Third-Party APIs**: These are not built into the browser by default. You have to retrieve their code in some way. Usually, they will have detailed documentation explaining how to use their services. An example is the Google Maps API, which you can use to display interactive maps on your website.
- **DOM**: The DOM stands for Document Object Model. It's a programming interface that lets you interact with HTML documents. With the DOM, you can add, modify, or delete elements on a webpage. The root of the DOM tree is the `html` element. It's the top-level container for all the content of an HTML document. All other nodes are descendants of this root node. Then, below the root node, we find other nodes in the hierarchy. A parent node is an element that contains other elements. A child node is an element that is contained within another element.
- `navigator` **Interface**: This provides information about the browser environment, such as the user agent string, the platform, and the version of the browser. A user agent string is a text string that identifies the browser and operating system being used.
- `window` **Interface**: This represents the browser window that contains the DOM document. It provides methods and properties for interacting with the browser window, such as resizing the window, opening new windows, and navigating to different URLs.

Working with the `querySelector()`, `querySelectorAll()` and `getElementById()` Methods

- `getElementById()` **Method**: This method is used to get an object that represents the HTML element with the specified `id`. Remember that ids must be unique in every HTML document, so this method will only return one Element object.

```
<div id="container"></div>
```

```
const container = document.getElementById("container");
```

- `querySelector()` **Method**: This method is used to get the first element in the HTML document that matches the CSS selector passed as an argument.

```
<section class="section"></section>
```

```
const section = document.querySelector(".section");
```

- `querySelectorAll()` **Method**: You can use this method to get a list of all the DOM elements that match a specific CSS selector.

```
<ul class="ingredients">
  <li>Sugar</li>
  <li>Milk</li>
  <li>Eggs</li>
</ul>
```

```
const ingredients = document.querySelectorAll('ul.ingredients li');
```

Working with the `innerText()`, `innerHTML()`, `createElement()` and `textContent()` Methods

- **`innerHTML` Property**: This is a property of the `Element` that is used to set or update parts of the HTML markup.

```
<div id="container">
  <!-- Add new elements here -->
</div>
```

```
const container = document.getElementById("container");
container.innerHTML = '<ul><li>Cheese</li><li>Tomato</li></ul>';
```

- **`createElement` Method**: This is used to create an HTML element.

```
const img = document.createElement("img");
```

- **`innerText`**: This represents the visible text content of the HTML element and its descendants.

```
<div id="container">
  <p>Hello, World!</p>
  <p>I'm learning JavaScript</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.innerText);
```

- **`textContent`**: This returns the plain text content of an element, including all the text within its descendants.

```
<div id="container">
  <p>Hello, World!</p>
  <p>I'm learning JavaScript</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.textContent);
```

Working with the `appendChild()` and `removeChild()` Methods

- **`appendChild()` Method**: This method is used to add a node to the end of the list of children of a specified parent node.

```
<ul id="desserts">
  <li>Cake</li>
  <li>Pie</li>
</ul>
```

```
const dessertsList = document.getElementById("desserts");
const listItem = document.createElement("li");

listItem.textContent = "Cookies";
dessertsList.appendChild(listItem);
```

- `removeChild()` **Method**: This method is used to remove a node from the DOM.

```
<section id="example-section">
  <h2>Example sub heading</h2>
  <p>first paragraph</p>
  <p>second paragraph</p>
</section>
```

```
const sectionEl = document.getElementById("example-section");
const lastParagraph = document.querySelector("#example-section p:last-of-
type");

sectionEl.removeChild(lastParagraph);
```

Work with the `setAttribute()` Method

- **Definition**: This method is used to set the attribute for a given element. If the attribute already exists, then the value is updated. Otherwise, a new attribute is added with a value.

```
<p id="para">I am a paragraph</p>
```

```
const para = document.getElementById("para");
para.setAttribute("class", "my-class");
```

Event Object

- **Definition**: The `Event` object is a payload that triggers when a user interacts with your web page in some way. These interactions can be anything from clicking on a button or focusing an input to shaking their mobile device. All `Event` objects will have the `type` property. This property reveals the type of event that triggered the payload, such as keydown or click. These values will correspond to the same values you might pass to `addEventListener`(), where you can capture and utilize the `Event` object.

`addEventListener()` and `removeEventListener()` Methods

- `addEventListener` **Method**: This method is used to listen for events. It takes two arguments: the event you want to listen for and a function that will be called when the event occurs. Some common examples of events would be click events, input events, and change events.

```
const btn = document.getElementById("btn");

btn.addEventListener("click", () => alert("You clicked the button"));
```

- `removeEventListener()` **Method**: This method is used to remove an event listener that was previously added to an element using the `addEventListener()` method. This is useful when you want to stop listening for a particular event on an element.

```
const bodyEl = document.querySelector("body");
const para = document.getElementById("para");
const btn = document.getElementById("btn");

let isBgColorGrey = true;

function toggleBgColor() {
  bodyEl.style.backgroundColor = isBgColorGrey ? "blue" : "grey";
  isBgColorGrey = !isBgColorGrey;
}

btn.addEventListener("click", toggleBgColor);

para.addEventListener("mouseover", () => {
  btn.removeEventListener("click", toggleBgColor);
});
```

- **Inline Event Handlers**: Inline event handlers are special attributes on an HTML element that are used to execute JavaScript code when an event occurs. In modern JavaScript, inline event handlers are not considered best practice. It is preferred to use the `addEventListener` method instead.

```
<button onclick="alert('Hello World!')">Show alert</button>
```

The Change Event

- **Definition**: The change event is a special event which is fired when the user modifies the value of certain input elements. Examples would include when a checkbox or a radio button is ticked. Or when the user makes a selection from something like a date picker or dropdown menu.

```
<label>
  Choose a programming language:
  <select class="language" name="language">
    <option value="">---Select One---</option>
    <option value="JavaScript">JavaScript</option>
    <option value="Python">Python</option>
    <option value="C++">C++</option>
  </select>
</label>

<p class="result"></p>
```

```
const selectEl = document.querySelector(".language");
const result = document.querySelector(".result");

selectEl.addEventListener("change", (e) => {
  result.textContent = `You enjoy programming in ${e.target.value}.`;
});
```

Event Bubbling

- **Definition**: Event bubbling, or propagation, refers to how an event "bubbles up" to parent objects when triggered.
- `stopPropagation()` **Method**: This method prevents further propagation for an event.

Event Delegation

- **Definition**: Event delegation is the process of listening to events that have bubbled up to a parent, rather than handling them directly on the element that triggered them.

DOMContentLoaded

- **Definition**: The `DOMContentLoaded` event is fired when everything in the HTML document has been loaded and parsed. If you have external stylesheets, or images, the `DOMContentLoaded` event will not wait for those to be loaded. It will only wait for the HTML to be loaded.

Working with `style` and `classList`

- `Element.style` **Property**: This property is a read-only property that represents the inline style of an element. You can use this property to get or set the style of an element.

```
const paraEl = document.getElementById("para");
paraEl.style.color = "red";
```

- `Element.classList` **Property**: This property is a read-only property that can be used to add, remove, or toggle classes on an element.

```
// Example adding a class
const paraEl = document.getElementById("para");
paraEl.classList.add("highlight");

// Example removing a class
paraEl.classList.remove("blue-background");

// Example toggling a class
const menu = document.getElementById("menu");
const toggleBtn = document.getElementById("toggle-btn");

toggleBtn.addEventListener("click", () => menu.classList.toggle("show"));
```

Working with the `setTimeout()` and `setInterval()` Methods

- `setTimeout()` **Method**: This method lets you delay an action for a specified time.

```
setTimeout(() => {
 console.log('This runs after 3 seconds');
}, 3000);
```

- `setInterval()` **Method**: This method keeps runs a piece of code repeatedly at a set interval. Since `setInterval()` keeps executing the provided function at the specified interval, you might want to stop it. For this, you have to use the `clearInterval()` method.

```
setInterval(() => {
 console.log('This runs every 2 seconds');
}, 2000);
```

```
// Example using clearInterval
const intervalID = setInterval(() => {
 console.log('This will stop after 5 seconds');
}, 1000);

setTimeout(() => {
 clearInterval(intervalID);
}, 5000);
```

The `requestAnimationFrame()` Method

- **Definition**: This method allows you to schedule the next step of your animation before the next screen repaint, resulting in a fluid and visually appealing experience. The next screen repaint refers to the moment when the browser refreshes the visual display of the web page. This happens multiple times per second, typically around 60 times (or 60 frames per second) on most displays.

```
function animate() {
 // Update the animation...
 // for example, move an element, change a style, and more.
 update();
 // Request the next frame
 requestAnimationFrame(animate);
}
```

Web Animations API

- **Definition**: The Web Animations API lets you create and control animations directly inside JavaScript.

```
const square = document.querySelector('#square');

const animation = square.animate(
 [{ transform: 'translateX(0px)' }, { transform: 'translateX(100px)' }],
 {
   duration: 2000, // makes animation lasts 2 seconds
   iterations: Infinity, // loops indefinitely
   direction: 'alternate', // moves back and forth
   easing: 'ease-in-out', // smooth easing
 }
);
```

The Canvas API

- **Definition**: The Canvas API is a powerful tool that lets you manipulate graphics right inside your JavaScript file. To work with the Canvas API, you first need to provide a `canvas` element in HTML. This element acts as a drawing surface you can manipulate with the instance methods and properties of the interfaces in the Canvas API. This API has interfaces like `HTMLCanvasElement`, `CanvasRenderingContext2D`, `CanvasGradient`, `CanvasPattern`, and `TextMetrics` which contain methods and properties you can use to create graphics in your JavaScript file.

```
<canvas id="my-canvas" width="400" height="400"></canvas>
```

```
const canvas = document.getElementById('my-canvas');

// Access the drawing context of the canvas.
```

```
// "2d" allows you to draw in two dimensions
const ctx = canvas.getContext('2d');

// Set the background color
ctx.fillStyle = 'crimson';

// Draw a rectangle
ctx.fillRect(1, 1, 150, 100);
```

Opening and Closing Dialogs and Modals with JavaScript

- **Modal and Dialog Definitions**: Dialogs let you display important information or actions to users. With the HTML built-in dialog element, you can easily create these dialogs (both modal and non-modal dialogs) in your web apps. A modal dialog is a type of dialog that forces the user to interact with it before they can access the rest of the application or webpage. In contrast, a non-modal dialog allows the user to continue interacting with other parts of the page or application even when the dialog is open. It doesn't prevent access to the rest of the content.
- `showModal()` **Method**: This method is used to open a modal.

```
<dialog id="my-modal">
   <p>This is a modal dialog.</p>
</dialog>
<button id="open-modal">Open Modal Dialog</button>
```

```
const dialog = document.getElementById('my-modal');
const openButton = document.getElementById('open-modal');

openButton.addEventListener('click', () => {
  dialog.showModal();
});
```

- `close()` **Method**: This method is used to close the modal.

```
<dialog id="my-modal">
   <p>This is a modal dialog.</p>
   <button id="close-modal">Close Modal</button>
</dialog>
<button id="open-modal">Open Modal Dialog</button>
```

```
const dialog = document.getElementById('my-modal');
const openButton = document.getElementById('open-modal');
const closeButton = document.getElementById('close-modal');

openButton.addEventListener('click', () => {
  dialog.show();
});

closeButton.addEventListener('click', () => {
  dialog.close();
});
```

--assignment--

Review the DOM Manipulation and Click Events with JavaScript topics and concepts.

Dynamic Programming Review

Introduction to Dynamic Programming

- **Definition**: Dynamic programming is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations.
- **Overlapping Subproblems**: The same smaller problems appear multiple times when solving the larger problem. Instead of recalculating these subproblems repeatedly, we store their solutions.
- **Optimal Substructure**: The optimal solution to the problem contains optimal solutions to its subproblems. This means we can build up the best solution by combining the best solutions to smaller parts.

Dynamic Programming Solutions

- **Memoization (Top-Down Approach)**: Memoization stores the results of expensive function calls and returns the cached result when the same inputs occur again.

```
def climb_stairs_memo(n, memo={}):
    """Dynamic programming with memoization"""
    # Check if we've already calculated this value
    if n in memo:
        return memo[n]  # Return cached result - O(1) lookup!

    # Base cases
    if n <= 2:
        return n

    # Calculate once and store in memo for future use
    memo[n] = climb_stairs_memo(n-1, memo) + climb_stairs_memo(n-2, memo)
    return memo[n]
```

- **Tabulation (Bottom-Up Approach)**: Tabulation builds the solution from the ground up, filling a table with solutions to subproblems.

```
def climb_stairs_tabulation(n):
    """Dynamic programming with tabulation"""
    if n <= 2:
        return n

    # Create array to store results for all steps from 0 to n
    dp = [0] * (n + 1)
    dp[1] = 1  # 1 way to reach step 1
    dp[2] = 2  # 2 ways to reach step 2

    # Build up the solution iteratively
    for i in range(3, n + 1):
        # Ways to reach step i = ways to reach (i-1) + ways to reach (i-2)
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

Real-World Applications Using Dynamic Programming

- **Route Optimization**: GPS systems use dynamic programming algorithms to find shortest paths between locations.
- **Text Processing**: Spell checkers and autocomplete features often rely on dynamic programming to calculate edit distances between words.
- **Financial Modeling**: Investment strategies and portfolio optimization frequently employ dynamic programming techniques.
- **Resource Allocation**: The knapsack problem and its variants appear in scheduling, budgeting, and resource management.

When to Use Dynamic Programming

You should consider using dynamic programming in the following scenarios:

- The problem can be broken down into overlapping subproblems.
- The problem exhibits optimal substructure.
- A naive recursive solution would involve repeated calculations.
- You need to optimize for time complexity at the cost of space complexity.

--assignment--

Review Dynamic Programming topics and concepts.

Error Handling Review

Common Errors in Python

- **SyntaxError**: The error Python raises when your code does not follow its syntax rules. For example, the code `print("Hello there"` will lead to a syntax error with the message, `SyntaxError: '(' was never closed`, because the code is missing a closing parenthesis.
- **NameError**: Python raises a `NameError` when you try to access a variable or function you have not defined. For instance, if you have the line `print(username)` in your code without having a `username` variable defined first, you will get a name error with the message `NameError: name 'username' is not defined`.
- **TypeError**: This is the error Python throws when you perform an operation on two or more incompatible data types. For example, if you try to add a string to a number, you'll get the error `TypeError: can only concatenate str (not "int") to str`.
- **IndexError**: You'll get an `IndexError` if you access an index that does not exist in a list or other sequences like tuple and string. For example, in a `Hello world` string, the index of the last character is `11`. If you go ahead and access a character this way, `greet = "hello world"; print(greet[12])`, you'll get an error with the message `IndexError: string index out of range`.
- **AttributeError**: Python raises this error when you try to use a method or property that does not exist in an object of that type. For example, calling `.append()` on a string like `"hello".append("!")` will lead to an error with the message `AttributeError: 'str' object has no attribute 'append'`.

Good Debugging Techniques in Python

- **Using the `print` function**: Inserting `print()` statements around various points in your code while debugging helps you see the values of variables and how your code flows.
- **Using Python's Built-in Debugger (`pdb`)**: Python provides a `pdb` module for debugging. It's a part of the Python's standard library, so it's always available to use. With `pdb`, you can set a trace with the `set_trace()` method so you can start stepping through the code and inspect variables in an interactive way.
- **Leveraging IDE Debugging Tools**: Many integrated development environments (IDEs) and code editors like Pycharm and VS Code offer debugging tools with breakpoints, step execution, variable

inspection, and other debugging features.

Exception Handling

- **`try...except`**: This is used to execute a block of code that might raise an exception. The `try` block is where you anticipate an error might occur, while the `except` block takes a specified exception and runs if that specified error is raised. Here's an example:

```
try:
  print(22 / 0)
except ZeroDivisionError:
  print('You can\'t divide by zero!')
  # You can't divide by zero!
```

  You can also chain multiple `except` blocks so you can handle more types of exceptions:

```
try:
  number = int(input('Enter a number: '))
  print(22 / number)
except ZeroDivisionError:
  print('You cannot divide by zero!')
  # You cannot divide by zero! prints when you enter 0
except ValueError:
  print('Please enter a valid number!')
  # Please enter a valid number! prints when you enter a string
```

- **`else` and `finally`**: These blocks extend `try...except`. If no exception occurs, the `else` block runs. The `finally` block always runs regardless of errors.

```
try:
  result = 100 / 4
except ZeroDivisionError:
  print('You cannot divide by zero!') # This will not run
else:
  print(f'Result is {result}') # Result is 25.0
finally:
  print('Execution complete!') # Execution complete!
```

- **Exception Object**: This lets you access the exception itself for better debugging and printing the direct error message. To access the exception object, you need to use the `as` keyword. Here's an example:

```
try:
    value = int('This will raise an error')
except ValueError as e:
    print(f'Caught an error: {e}')
    # Caught an error: invalid literal for int() with base 10: 'This will
raise an error'
```

- **The `raise` Statement**: This allows you to manually raise an exception. You can use it to throw an exception when a certain condition is met. Here's an example:

```
def divide(a, b):
    if b == 0:
```

```
            raise ZeroDivisionError('You cannot divide by zero')
        return a / b
```

Exception Signaling

The `raise` statement is also useful when you create your own custom exceptions, as you can use it to throw an exception with a custom message. Here's an example of that:

```
class InvalidCredentialsError(Exception):
    def __init__(self, message="Invalid username or password"):
        self.message = message
        super().__init__(self.message)

def login(username, password):
    stored_username = "admin"
    stored_password = "password123"

    if username != stored_username or password != stored_password:
        raise InvalidCredentialsError()

    return f"Welcome, {username}!"
```

Here's a how you can use the `login` function from the `InvalidCredentialsError` exception:

```
# failed login attempt
try:
    message = login("user", "wrongpassword")
    print(message)
except InvalidCredentialsError as e:
    print(f"Login failed: {e}")

# successful login attempt
try:
    message = login("admin", "password123")
    print(message)
except InvalidCredentialsError as e:
    # This block is not executed because the login was successful
    print(f"Login failed: {e}")
else:
    # The else block runs if the 'try' block completes without an exception
    print(message)
```

The `raise` statement can also be used with the `from` keyword to chain exceptions, showing the relationship between different errors:

```
def parse_config(filename):
  try:
      with open(filename, 'r') as file:
          data = file.read()
          return int(data)
  except FileNotFoundError:
      raise ValueError('Configuration file is missing') from None
  except ValueError as e:
      raise ValueError('Invalid configuration format') from e

config = parse_config('config.txt')
```

--assignment--

Review the Error Handling topics and concepts.

Form Validation with JavaScript Review

Validating Forms with JavaScript

- **Constraint Validation API**: Certain HTML elements, such as the `textarea` and `input` elements, expose a constraint validation API. This API allows you to assert that the user's provided value for that element passes any HTML-level validation you have written, such as minimum length or pattern matching.
- **`checkValidity()` method**: This method returns `true` if the element matches all HTML validation (based on its attributes), and `false` if it fails.

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  if (!e.target.checkValidity()) {
    e.target.setCustomValidity("You must use a .com email.")
  }
})
```

- **`reportValidity()` Method**: This method tells the browser that the `input` is invalid.

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  if (!e.target.checkValidity()) {
    e.target.reportValidity();
  }
})
```

- **`validity` Property**: This property is used to get or set the validity state of form controls (like `<input>`, `<select>`, etc.) and provides information about whether the user input meets the constraints defined for that element (e.g., `required` fields, pattern constraints, maximum length, etc.).

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  console.log(e.target.validity);
})
```

- **`patternMismatch` Property**: This will be `true` if the value doesn't match the specified regular expression pattern.

`preventDefault()` Method

- **Definition**: Every event that triggers in the DOM has some sort of default behavior. The click event on a checkbox toggles the state of that checkbox, by default. Pressing the space bar on a focused button activates the button. The `preventDefault()` method on these `Event` objects stops that behavior from happening.

```
button.addEventListener('click', (event) => {
  // Prevent the default button click behavior
  event.preventDefault();
  alert('Button click prevented!');
});
```

Submitting Forms

- **Definition**: There are three ways a form can be submitted. The first is when the user clicks a button in the form which has the `type` attribute set to `submit`. The second is when the user presses the `Enter` key on any editable `input` field in the form. The third is through a JavaScript call to the `requestSubmit()` or `submit()` methods of the `form` element.
- **`action` Attribute**: The `action` attribute should contain either a URL or a relative path for the current domain. This value determines where the form attempts to send data - if you do not set an `action` attribute, the form will send data to the current page's URL.

```
<form action="https://freecodecamp.org">
  <input
    type="number"
    id="input"
    placeholder="Enter a number"
    name="number"
  />
  <button type="submit">Submit</button>
</form>
```

- **`method` Attribute**: This attribute accepts a standard `HTTP` method, such as `GET` or `POST`, and uses that method when making the request to the action URL. When a method is not set, the form will default to a `GET` request. The data in the form will be URL encoded as `name=value` pairs and appended to the action URL as query parameters.

```
<form action="/data" method="POST">
  <input
    type="number"
    id="input"
    placeholder="Enter a number"
    name="number"
  />
  <button type="submit">Submit</button>
</form>
```

- **`enctype` Attribute**: The `form` element accepts an `enctype` attribute, which represents the encoding type to use for the data. This attribute only accepts three values: `application/x-www-form-urlencoded` (which is the default, sending the data as a URL-encoded form body), `text/plain` (which sends the data in plaintext form, in `name=value` pairs separated by new lines), or `multipart/form-data`, which is specifically for handling forms with a file upload.

--assignment--

Review the Form Validation with JavaScript topics and concepts.

Front End Libraries Review

JavaScript Libraries and Frameworks

- JavaScript libraries and frameworks offer quick solutions to common problems and speed up development by providing pre-built code.
- Libraries are generally more focused on providing solutions to specific tasks, such as manipulating the DOM, handling events, or managing AJAX requests.
- A couple of examples of JavaScript libraries are jQuery and React.
- Frameworks, on the other hand, provide a more defined structure for building applications. They often come with a set of rules and conventions that developers need to follow.
- Examples of frameworks include Angular and Next.js, a meta framework for React.
- **Single-page applications** (SPAs) are web applications that load a single HTML page and dynamically update that page as the user interacts with the application without reloading the entire page.
- SPAs use JavaScript to manage the application's state and render content. This is often done using frameworks which provide great tools for building complex user interfaces.
- Some issues surrounding SPAs include:
    - Screen readers struggle with dynamically updated content.
    - The URL does not change when the user navigates within the application, which can make it difficult to bookmark, backtrack or share specific pages.
    - Initial load time can be slow if the application is large as all the assets need to be loaded upfront.

React

- React is a popular JavaScript library for building user interfaces and web applications.
- A core concept of React is the creation of reusable UI components that can update and render independently as data changes.
- React allows developers to describe how the UI should look like based on the application state. React then updates and renders the right components when the data or the state changes.

React Components

- Components are the building blocks of React applications that allow developers to break down complex user interfaces into smaller, manageable pieces.
- The UI is described using JSX, an extension of JavaScript syntax, that allows developers to write HTML-like code within JavaScript.
- Components are basically JS functions or classes that return a piece of UI.

Here is an example of a simple React component that renders a greeting message:

```
function Greeting() {
  const name = 'Anna';
  return <h1>Welcome, {name}!</h1>;
}
```

To use the component, you can simply call:

```
  <Greeting />
```

Importing and Exporting React components

- React components can be exported from one file and imported into another file.
- Let's say you have a component named `City` in a file named `City.js`. You can export the component using the `export` keyword:

```
// City.js
function City() {
  return <p>New York</p>;
}

export default City;
```

- To import the `City` component into another file, you can use the `import` keyword:

```
// App.js
import City from './City';

function App() {
  return (
    <div>
      <h1>My favorite city is:</h1>
      <City />
    </div>
  );
}
```

- The `default` keyword is used as it is the default export from the `City.js` file.
- You can also choose to export the component on the same line as the component definition like this:

```
export default function City() {
  return <p>New York</p>;
}
```

Setting up a React project using Vite

- Project setup tools and CLIs provide a quick & easy way to start new projects, allowing developers to focus on writing code rather than dealing with configuration.
- Vite, a popular project setup tool and can be used with React.
- To create a new project with Vite, you can use the following command in your terminal:

```
npm create vite@latest my-react-app -- --template react
```

This command creates a new React project named `my-react-app` using Vite's React template. In the project directory, you will see a `package.json` file with the project dependencies and commands listed in it.

- To run the project, navigate to the project directory and run the following commands:

```
cd my-react-app # path to the project directory
npm install # installs the dependencies listed in the package.json file
```

- Once the dependencies are installed, you should notice a new folder in your project called `node_modules`.

- The `node_modules` folder is where all the packages and libraries required by your project are stored.
- To run your project, use the following command:

```
npm run dev
```

- After that, open your browser and navigate to `http://localhost:5173` to see your React application running.
- To actually see the code for the starter template, you can go into your project inside the `src` folder and you should see the `App.jsx` file.

Passing props in React components

- In React, props (short for properties) are a way to pass data from a parent component to a child component. This mechanism is needed to create reusable and dynamic UI elements.
- Props can be any JavaScript value. To pass props from a parent to a child component, you add the props as attributes when you use the child component in the parent's JSX. Here's a simple example:

```
// Parent component
function Parent() {
  const name = 'Anna';
  return <Child name={name} />;
}

// Child component
function Child(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

You can pass multiple props using the spread operator `(...)`, after converting them to an object. Here's an example:

```
// Parent component
function Parent() {
  const person = {
    name: 'Anna',
    age: 25,
    city: 'New York'
  };
  return <Child {...person} />;
}
```

In this code, the spread operator `{...person}` converts the person object into individual props that are passed to the Child component.

Conditional rendering in React

- Conditional rendering in React allows you to create dynamic user interfaces. It is used to show different content based on certain conditions or states within your application.
- There are several ways to conditionally render content in React. One common approach is to use the ternary operator. Here's an example:

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
```

```
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please log in</h1>}
    </div>
  );
}
```

- Another way to conditionally render content is to use the logical AND (&&) operator. This is useful when you want to render content only if a certain condition is met. Here's an example:

```
function Greeting({ user }) {
  return (
    <div>
      {user && <h1>Welcome, {user.name}!</h1>}
    </div>
  );
}
```

In the code above, the `h1` element is only rendered if the user object is truthy.

You can also use a direct `if` statement this way:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please sign in</h1>;
}
```

Rendering lists in React

- Rendering lists in React is a common task when building user interfaces.
- Lists can be rendered using the JS array `map()` method to iterate over an array of items and return a new array of JSX elements.
- For example, if you have an array of names that you want to render as a list, you can do the following:

```
function NameList({ names }) {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={`${name}-${index}`}>{name}</li>
      ))}
    </ul>
  );
}
```

- Always remember to provide a unique key for each list item to help React manage the updating and rendering roles. With these techniques, you can create flexible, efficient, and dynamic lists in your React applications.

Inline styles in React

- Inline styles in React allow you to apply CSS styles directly to JSX elements using JavaScript objects.
- To apply inline styles in React, you can use the style attribute on JSX elements. The style attribute takes an object where the keys are CSS properties in camelCase and the values are the corresponding values. Here's an example:

```
function Greeting() {
  return (
    <h1
      style={{ color: 'blue', fontSize: '24px', backgroundColor: 'lightgray'
}}
    >
      Hello, world!
    </h1>
  );
}

export default Greeting;
```

You can also extract the styles into a separate object and reference it in the `style` attribute this way:

```
function Greeting() {

  const styles = {
    color: 'blue',
    fontSize: '24px',
    backgroundColor: 'lightgray'
  };

  return <h1 style={styles}>Hello, world!</h1>;
}

export default Greeting;
```

- Inline styles support dynamic styling by allowing you to conditionally apply styles based on props or state. Here is an example of how you can conditionally apply styles based on a prop:

```
function Greeting({ isImportant }) {

  const styles = {
    color: isImportant ? 'red' : 'black',
    fontSize: isImportant ? '24px' : '16px'
  };

  return <h1 style={styles}>Hello, world!</h1>;
}

export default Greeting;
```

- In the code above, the `color` and `fontSize` styles are conditionally set based on the `isImportant` prop.

Working with Events in React

- **Synthetic Event System**: This is React's way of handling events. It serves as a wrapper around the native events like the `click`, `keydown`, and `submit` events. Event handlers in React use the camel casing naming convention. (Ex. `onClick`, `onSubmit`, etc)

Here is an example of using the `onClick` attribute for a `button` element in React:

```
function handleClick() {
  console.log("Button clicked!");
}
```

```
<button onClick={handleClick}>Click Me</button>;
```

In React, event handler functions usually start with the prefix `handle` to indicate they are responsible for handling events, like `handleClick` or `handleSubmit`.

When a user action triggers an event, React passes a Synthetic Event object to your handler. This object behaves much like the native Event object in vanilla JavaScript, providing properties like `type`, `target`, and `currentTarget`.

To prevent default behaviors like browser refresh during an `onSubmit` event, for example, you can call the `preventDefault()` method:

```
function handleSubmit(event) {
  event.preventDefault();
  console.log("Form submitted!");
}

<form onSubmit={handleSubmit}>
  <input type="text" />
  <button>Submit</button>
</form>;
```

You can also wrap a handler function in an arrow function like this:

```
function handleDelete(id) {
  console.log("Deleting item:", id);
}

<button onClick={() => handleDelete(1)}>Delete Item</button>;
```

Working with State and the `useState` Hook

- **Definition for state**: In React, state is an object that contains data for a component. When the state updates the component will re-render. React treats state as immutable, meaning you should not modify it directly.
- **`useState()` Hook**: The `useState` hook is a function that lets you declare state variables in functional components. Here is a basic syntax:

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

In the state variable you have the following:

- `stateVariable` holds the current state value
- `setStateFunction` (the setter function) updates the state variable
- `initialValue` sets the initial state

Here is a complete example for a `Counter` component:

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
```

```
      <div>
        <h2>{count}</h2>

        <button onClick={() => setCount(count - 1)}>Decrement</button>
        <button onClick={() => setCount(count + 1)}>Increment</button>
      </div>
    );
}

export default Counter;
```

Rendering and React Components

- **Definition**: In React, rendering is the process by which components appear in the user interface (UI), usually the browser. The rendering process consists of three stages: trigger, render, and commit.

The trigger stage occurs when React detects that something has changed and the user interface (UI) might need to be updated. This change is often due to an update in state or props.

Once the trigger happens, React enters the render stage. Here, React re-evaluates your components and figures out what to display. To do this, React uses a lightweight copy of the "real" DOM called the virtual DOM. With the virtual DOM, React can quickly check what needs to change in the component.

The commit stage is where React takes the prepared changes from the virtual DOM and applies them to the real DOM. In other words, this is the stage where you see the final result on the screen.

Updating Objects and Arrays in State

- **Updating Objects in State**: If you need to update an object in state, then you should make a new object or copy an existing object first, then set the state for that new object. Any object put into state should be considered as read-only. Here is an example of setting a user's name, age and city. The `handleChange` function is used to handle updates to the user's information:

```
import { useState } from "react";

function Profile() {
  const [user, setUser] = useState({ name: "John Doe", age: 31, city: "LA"
});

  const handleChange = (e) => {
    const { name, value } = e.target;

    setUser((prevUser) => ({...prevUser, [name]: value}));
  };

  return (
    <div>
      <h1>User Profile</h1>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
      <p>City: {user.city}</p>

      <h2>Update User Age </h2>
      <input type="number" name="age" value={user.age} onChange=
{handleChange} />

      <h2>Update User Name </h2>
      <input type="text" name="name" value={user.name} onChange=
{handleChange} />

      <h2>Update User City </h2>
      <input type="text" name="city" value={user.city} onChange=
```

```
{handleChange} />
    </div>
  );
}

export default Profile;
```

- **Updating Arrays in State**: When updating arrays in state, it is important not to directly modify the array using methods like `push()` or `pop()`. Instead you should create a new array when updating state:

```
const addItem = () => {
  const newItem = {
    id: items.length + 1,
    name: `Item ${items.length + 1}`,
  };

  // Creates a new array
  setItems((prevItems) => [...prevItems, newItem]);
};
```

If you want to remove items from an array, you should use the `filter()` method, which returns a new array after filtering out whatever you want to remove:

```
const removeItem = (id) => {
  setItems((prevItems) => prevItems.filter((item) => item.id !== id));
};
```

Referencing Values Using Refs

- **`ref` Attribute**: You can access a DOM node in React by using the `ref` attribute. Here is an example to showcase a `ref` to focus an `input` element. The `current` property is used to access the current value of that `ref`:

```
import { useRef } from "react";

const Focus = () => {
  const inputRef = useRef(null);

  const handleFocus = () => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Enter text" />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );
};

export default Focus;
```

Working with the `useEffect` Hook

- `useEffect()` **Hook**: In React, an effect is anything that happens outside the component rendering process. That is, anything React does not handle directly as part of rendering the UI. Common examples include fetching data, updating the browser tab's title, reading from or writing to the browser's local storage, getting the user's location, and much more. These operations interact with the outside world and are known as side effects. React provides the `useEffect` hook to let you handle those side effects. `useEffect` lets you run a function after the component renders or updates.

```
import { useEffect } from "react";

useEffect(() => {
 // Your side effect logic (usually a function) goes here
}, [dependencies]);
```

The effect function runs after the component renders, while the optional `dependencies` argument controls when the effect runs.

Note that `dependencies` can be an array of "reactive values" (state, props, functions, variables, and so on), an empty array, or omitted entirely. Here's how all of those options control how `useEffect` works:

- If `dependencies` is an array that includes one or more reactive values, the effect will run whenever they change.
- If `dependencies` is an empty array, `useEffect` runs only once when the component first renders.
- If you omit `dependencies`, the effect runs every time the component renders or updates.

How to Create Custom Hooks

- **Custom Hooks**: A custom hook allows you to extract reusable logic from components, such as data fetching, state management, toggling, and side effects like tracking online status. In React, all built-in hooks start with the word `use`, so your custom hook should follow the same convention.

Here is an example of creating a `useDebounce` hook:

```
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

export { useDebounce };
```

Working with Forms in React

- **Controlled Inputs**: This is when you store the input field value in state and update it through `onChange` events. This gives you complete control over the form data and allows instant validation and conditional rendering.

```
import { useState } from "react";

function App() {
  const [name, setName] = useState("");

  const handleChange = (e) => {
    setName(e.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(name);
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <label htmlFor="name">Your name</label> <br />
        <input value={name} id="name" onChange={handleChange} type="text" />
        <button type="submit">Submit</button>
      </form>
    </>
  );
}

export default App;
```

- **Uncontrolled Inputs**: Instead of handling the inputs through the `useState` hook, uncontrolled inputs in HTML maintain their own internal state with the help of the DOM. Since the DOM controls the input values, you need to pull in the values of the input fields with a `ref`.

```
import { useRef } from "react";

function App() {
 const nameRef = useRef();

 const handleSubmit = (e) => {
   e.preventDefault();
   console.log(nameRef.current.value);
 };

 return (
   <form onSubmit={handleSubmit}>
     <label htmlFor="name">Your</label>{" "}
     <input type="text" ref={nameRef} id="name" />
     <button type="submit">Submit</button>
   </form>
 );
}

export default App;
```

Working with the `useActionState` Hook

- **Server Actions**: These are functions that run on the server to allow form handling right on the server without the need for API endpoints. Here is an example from a Next.js application:

```
"use server";
```

```
async function submitForm(formData) {
 const name = formData.get("name");
 return { message: `Hello, ${name}!` };
}
```

The `"use server"` directive marks the function as a server action.

- **`useActionState` Hook**: This hook updates state based on the outcome of a form submission. Here's the basic syntax of the `useActionState` hook:

```
const [state, action, isPending] = useActionState(actionFunction,
initialState, permalink);
```

- `state` is the current state the action returns.
- `action` is the function that triggers the server action.
- `isPending` is a boolean that indicates whether the action is currently running or not.
- `actionFunction` parameter is the server action itself.
- `initialState` is the parameter that represents the starting point for the state before the action runs.
- `permalink` is an optional string that contains the unique page URL the form modifies.

Data Fetching in React

- **Options For Fetching Data**: There are many different ways to fetch data in React. You can use the native Fetch API, or third party tools like Axios or SWR.
- **Commonly Used State Variables When Fetching Data**: Regardless of which way you choose to fetch your data in React, there are some pieces of state you will need to keep track of. The first is the data itself. The second will track whether the data is still being fetched. The third is a state variable that will capture any errors that might occur during the data fetching process.

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
```

Since data fetching is a side effect, it's best to use the `Fetch API` inside of a `useEffect` hook.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/posts");

      if (!res.ok) {
        throw new Error("Network response was not ok");
      }

      const data = await res.json();
      setData(data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);
```

Then you can render a loading message if the data fetching is not complete, an error message if there was an error fetching the data, or the results.

```
if (loading) {
  return <p>Loading...</p>;
}

if (error) {
  return <p>{error.message}</p>;
}

return (
  <ul>
    {data.map((post) => (
      <li key={post.id}>{post.title}</li>
    ))}
  </ul>
);
```

If you want to use Axios, you need to install and import it:

```
npm i axios
```

```
import axios from "axios";
```

Then you can fetch the data using `axios.get`:

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await axios.get(
        "https://jsonplaceholder.typicode.com/users"
      );
      setData(res.data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);
```

To fetch data using the `useSWR` hook, you need to first install and import it.

```
npm i swr
```

```
import useSWR from "swr";
```

Here is how you can use the hook to fetch data:

```
import useSWR from "swr";

const fetcher = (url) => fetch(url).then((res) => res.json());

const FetchTodos = () => {
 const { data, error } = useSWR(
   "https://jsonplaceholder.typicode.com/todos",
   fetcher
 );

 if (!data) {
   return <h2>Loading...</h2>;
 }
 if (error) {
   return <h2>Error: {error.message}</h2>;
 }

 return (
   <>
     <h2>Todos</h2>
     <div>
       {data.map((todo) => (
         <h3 key={todo.id}>{todo.title}</h3>
       ))}
     </div>
   </>
 );
};

export default FetchTodos;
```

Working with the `useOptimistic` Hook

- **`useOptimistic` Hook**: This hook is used to keep UIs responsive while waiting for an async action to complete in the background. It helps manage "optimistic updates" in the UI, a strategy in which you provide immediate updates to the UI based on the expected outcome of an action, like waiting for a server response.

Here is the basic syntax:

```
const [optimisticState, addOptimistic] = useOptimistic(actualState,
updateFunction);
```

- `optimisticState` is the temporary state that updates right away for a better user experience.
- `addOptimistic` is the function that applies the optimistic update before the actual state changes.
- `actualState` is the real state value that comes from the result of an action, like fetching data from a server.
- `updateFunction` is the function that determines how the optimistic state should update when called.

Here is an example of using the `useOptimistic` hook in a `TaskList` component:

```
"use client";

import { useOptimistic } from "react";
```

```
export default function TaskList({ tasks, addTask }) {
  const [optimisticTasks, addOptimisticTask] = useOptimistic(
    tasks,
    (state, newTask) => [...state, { text: newTask, pending: true }]
  );

  async function handleSubmit(e) {
    e.preventDefault();
    const formData = new FormData(e.target);

    addOptimisticTask(formData.get("task"));

    addTask(formData);
    e.target.reset();
  }

  return <>{/* UI */}</>;
}
```

- `startTransition`: This is used to render part of the UI and mark a state update as a non-urgent transition. This allows the UI to be responsive during expensive updates. Here is the basic syntax:

```
startTransition(action)
```

The `action` performs a state update or triggers some transition-related logic. This ensures that urgent UI updates (like typing or clicking) are not blocked.

Working with the `useMemo` Hook

- **Memoization**: This is an optimization technique in which the result of expensive function calls are cached (remembered) based on specific arguments. When the same arguments are provided again, the cached result is returned instead of re-computing the function.
- `useMemo` **Hook**: This hook is used to memoize computed values. Here is an example of memoizing the result of sorting a large array. The `expensiveSortFunction` will only run when `largeArray` changes:

```
const memoizedSortedArray = useMemo(
  () => expensiveSortFunction(largeArray),
  [largeArray]
);
```

Working with the `useCallback` Hook

- `useCallback` **Hook**: This is used to memoize function references.

```
const handleClick = useCallback(() => {
 // code goes here
}, [dependency]);
```

- `React.memo`: This is used to memoize a component to prevent it from unnecessary re-renders when its prop has not changed.

```
const MemoizedComponent = React.memo(({ prop }) => {
 return (
```

```
    <>
      {/* Presentation */}
    </>
  )
});
```

Dependency Management Tools

- **Dependency Definition**: In software, a dependency is where one component or module in an application relies on another to function properly. Dependencies are common in software applications because they allow developers to use pre-built functions or tools created by others. The two core dependencies needed for a React project will be the `react` and `react-dom` packages:

```
"dependencies": {
  "react": "^18.3.1",
  "react-dom": "^18.3.1"
}
```

- **Package Manager Definition**: To manage software dependencies in a project, you will need to use a package manager. A package manager is a tool used for installation, updates and removal of dependencies. Many popular programming languages like JavaScript, Python, Ruby and Java, all use package managers. Popular package managers for JavaScript include npm, Yarn and pnpm.
- **`package.json` File**: This is a key configuration file in projects that contains metadata about your project, including its name, version, and dependencies. It also defines scripts, licensing information, and other settings that help manage the project and its dependencies.
- **`package-lock.json` File**: This file will lock down the exact versions of all packages that your project is using. When you update a package, then the new versions will be updated in the lock file as well.
- **`node_modules` Folder**: This folder contains the actual code for the dependencies listed in your `package.json` file, including both your project's direct dependencies and any dependencies of those dependencies.
- **Dev Dependencies**: These are packages that are only used for development and not in production. An example of this would be a testing library like Jest. You would install Jest as a dev dependency because it is needed for testing your application locally but not needed to have the application run in production.

```
"devDependencies": {
  "@eslint/js": "^9.17.0",
  "@types/react": "^18.3.18",
  "@types/react-dom": "^18.3.5",
  "@vitejs/plugin-react": "^4.3.4",
  "eslint": "^9.17.0",
  "eslint-plugin-react": "^7.37.2",
  "eslint-plugin-react-hooks": "^5.0.0",
  "eslint-plugin-react-refresh": "^0.4.16",
  "globals": "^15.14.0",
  "vite": "^6.0.5"
}
```

React Router

- **Introduction**: React Router is a third party library that allows you to add routing to your React applications. To begin, you will need to install React Router in an existing React project like this:

```
npm i react-router
```

Then inside of the `main.jsx` or `index.jsx` file, you will need to setup the route structure like this:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router";
import App from "./App.jsx";

import "./index.css";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<App />} />
      </Routes>
    </BrowserRouter>
  </StrictMode>
);
```

The `path` and `element` are used to couple the URL and UI components together. In this case, we are setting up a route for the homepage that points to the `App` component.

- **Multiple Views and Route Setup**: It is common in larger applications to have multiple views and routes setup like this:

```
<Routes>
  <Route index element={<Home />} />
  <Route path="about" element={<About />} />

  <Route path="products">
    <Route index element={<ProductsHome />} />
    <Route path=":category" element={<Category />} />
    <Route path=":category/:productId" element={<ProductDetail />} />
    <Route path="trending" element={<Trending />} />
  </Route>
</Routes>
```

The `index` prop in these examples is meant to represent the default route for a given path segment. So the `Home` component will be shown at the root `/` path while the `ProductsHome` component will be shown at the `/products` path.

- **Nesting Routes**: You can nest routes inside other routes which results in the path of the child route being appended to the parent route's path.

```
<Route path="products">
  <Route path="trending" element={<Trending />} />
</Route>
```

In the example above, the path for the trending products will be products/trending.

- **Dynamic Segments**: A dynamic segment is where any part of the URL path is dynamic.

```
<Route path=":category" element={<Category />} />
```

In this example we have a dynamic segment called `category`. When a user navigates to a URL like products/brass-instruments, then the view will change to the `Category` component and you can

dynamically fetch the appropriate data based on the segment.

- `useParams` **Hook**: This hook is used to access the dynamic parameters from a URL path.

```
import { useParams } from "react-router";

export default function Category() {
  let params = useParams();
   {/* Accessing the category param: params.category */}
   {/* rest of code goes here */}
}
```

React Frameworks

- **Introduction**: React frameworks provide features like routing, image optimizations, data fetching, authentication and more. This means that you might not need to set up separate frontend and backend applications for certain use cases. Examples of React Frameworks include Next.js and Remix.
- **Next.js Routing**: This routing system includes support for dynamic routes, parallel routes, route handlers, redirects, internalization and more.

Here is an example of creating a custom request handler:

```
export async function GET() {
  const res = await fetch("https://example-api.com");
  const data = await res.json();

  return Response.json({ data });
}
```

- **Next.js Image Optimization**: The `Image` component extends the native HTML `img` element and allows for faster page loads and size optimizations. This means that images will only load when they enter the viewport and the `Image` component will automatically serve correctly sized images for each device.

```
import Image from "next/image";

export default function Page() {
  return (
    <Image src="link-to-image-goes-here" alt="descriptive-title-goes-here" />
  );
}
```

Prop Drilling

- **Definition**: Prop drilling is the process of passing props from a parent component to deeply nested child components, even when some of the child components don't need the props.

State Management

- **Context API**: Context refers to when a parent component makes information available to child components without needing to pass it explicitly through props. `createContext` is used to create a context object which represent the context that other components will read. The `Provider` is used to supply context values to the child components.

```
```

```
import { useState, createContext } from "react";

const CounterContext = createContext();

const CounterProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  return (
    <CounterContext.Provider value={{ count, setCount }}>
      {children}
    </CounterContext.Provider>
  );
};

export { CounterContext, CounterProvider };
```

- **Redux**: Redux handles state management by providing a central store and strict control over state updates. It uses a predictable pattern with actions, reducers, and middleware. Actions are payloads of information that send data from your application to the Redux store, often triggered by user interactions. Reducers are functions that specify how the state should change in response to those actions, ensuring the state is updated in an immutable way. Middleware, on the other hand, acts as a bridge between the action dispatching and the reducer, allowing you to extend Redux's functionality (e.g., logging, handling async operations) without modifying the core flow.

- **Zustand**: This state management solution is ideal for small to medium-scale applications. It works by using a `useStore` hook to access state directly in components and pages. This lets you modify and access data without needing actions, reducers, or a provider.

Debugging React Components Using the React DevTools

- **React Developer Tools**: This is a browser extension you can use in Chrome, Firefox and Edge to inspect React components and identify performance issues. For Safari, you will need to install the `react-devtools` npm package. After installing React DevTools and opening a React app in the browser, open the browser developer tools to access the two extra tabs provided for debugging React – Components and Profiler.
- **Components Tab**: This tab displays each component for you in a tree view format. Here are some things you can do in this tab:
  - view the app's component hierarchy
  - check and modify props, states, and context values in real time
  - check the source code for each selected component
  - log the component data to the console
  - inspect the DOM elements for the component
- **Profiler Tab**: This tab helps you analyze component performance. You can record component performance so you can identify unnecessary re-renders, view commit durations, and subsequently optimize slow components.

React Server Components

- **Definition**: React Server Components are React components that render exclusively on the server, sending only the final HTML to the client. This means those components can directly access server-side resources and dramatically reduce the amount of JavaScript sent to the browser.

Differences Between Real and Perceived Performance

- **Perceived Performance**: This is how users perceive the performance of a website. It's how they evaluate it in terms of responsiveness and reliability. This is a subjective measurement, so it's hard to quantify it, but it's very important, since the user experience determines the success or failure of a website.

- **Real Performance**: This is the objective and measurable performance of the website. It's measured using metrics like page load time, server response time, and rendering time. These measurements are influenced by multiple factors related to the network and to the code itself.

Techniques for Improving Perceived Performance

- **Lazy Loading**: This technique reduces the initial load time as much as possible by loading non-essential resources in the background.
- **Minimize Font Delays**: If your website has custom fonts, you should also try to minimize font loading delays, since this may result in flickering or in showing the fallback font while the custom font is being loaded. A suggestion for this is using a fallback font that is similar to the custom font, so in case this happens, the change will be more subtle.
- **Use of Loading Indicators**: Showing a loading indicator for a long-running process as soon as the user clicks on an element can help the user feel connected and engaged with the process, making the wait time feel shorter.

Core Performance Concepts

- **Source order**: This refers to the way HTML elements are structured in the document. This determines what loads first and can significantly impact performance and accessibility.

Some best practices for source order include:

- Placing critical content such as headings, navigation or main text higher in the HTML structure.
- Deferring non-essential scripts such as ones for analytics, or third-party widgets, so they don't block rendering.
- Using progressive enhancement, to ensure the core experience works even before styles and scripts load. Progressive enhancement is a way of building websites and applications based on the idea that you should make your page work with HTML first.

Here is an example of good source order, using the best practices we just went through:

```
<h1>Welcome to FastSite!</h1>
<p>Critical information loads first.</p>
<script src="slow-script.js" defer></script>
```

- **Critical Rendering Path**: This is the sequence of steps the browser follows to convert code into pixels on the screen.
- **Latency**: This is the time it takes for a request to travel between the browser and the server. So in other words, high latency equals slow pages.

Some ways of reducing latency include:

- Using CDNs, or Content Delivery Networks, to serve files from closer locations.
- Enabling compression using things such as Gzip to reduce file sizes.
- Optimizing images and using lazy loading.

```
<img src="placeholder.jpg" data-src="real-image.jpg" loading="lazy">
```

Improving INP

- **Definition**: INP (Interaction to Next Paint) assesses a page's overall responsiveness by measuring the time from when a user interacts, like a click or key press, to the next time the browser updates the display. A lower INP indicates a more responsive page.

Here are some ways to improve INP:

- Reduce main thread work by breaking up long JavaScript tasks.
- Use `requestIdleCallback()` for non-critical scripts. This will queue a function to be called during a browser's idle periods.
- Defer or lazy-load heavy assets which were covered earlier.
- Optimize event handlers. If these handlers run too frequently or perform heavy operations, they can slow down the page and increase INP. The solution for this is debouncing. Debouncing ensures that the function only runs after the user stops typing for a short delay - so for example 300ms. This prevents unnecessary calculations and improves performance.

How Rendering Works in the Browser

- **How Rendering works**: First the browser parses the HTML and builds the DOM. Next, the browser processes the CSS, constructing the CSS Object Model, or CSSOM. This is another tree structure that dictates how elements should be styled. Finally, the browser paints the pixels to the screen, rendering each element based on the calculated styles and layout. In complex pages, this might involve multiple layers that are composited together to form the final visual output.

How Performance Impacts Sustainability

- **Background Information**: The internet accounts for around 2% of global carbon emissions—that's the same as the airline industry! Every byte transferred requires electricity, from data centers to user devices. Larger files and inefficient scripts mean more power consumption. A high-performance website isn't just faster, it also reduces unnecessary processing and energy use.

Ways to Reduce Page Loading Times

- **Optimize Media Assets**: Large images and videos are common culprits for slow load times. By optimizing these assets, you can significantly speed up your site. This includes things like compressing images, using modern formats like WebP and using lazy loading for assets.
- **Leverage Browser Caching**: Caching allows browsers to store parts of your website locally, reducing load times for returning visitors.
- **Minify and Compress Files**: Reducing the size of your files can lead to quicker downloads. This includes reducing the size of transmitted files and minifying CSS and JavaScript files.

Improving "time to usable"

- **Definition**: "time to usable" is the interval from when a user requests a page to when they can meaningfully interact with it. To improve "time to usable" you can lazy load your asset or minimize render-blocking resources.

Key Metrics for Measuring Performance

- **First Contentful Paint or FCP**: It measures how quickly the first piece of content—text or image—appears on the screen. A good FCP is regarded as a time below 1.8 seconds, and a poor FCP is above 3 seconds. You can check your FCP using Chrome DevTools and checking the performance tab.
- **Total Blocking Time**: This shows how long the main thread is blocked by heavy JavaScript tasks. If Total Blocking Time (TBT) is high, users experience sluggish interactions. To improve TBT, break up long tasks and defer non-essential scripts.
- **Bounce Rate**: This is the percentage of visitors who leave without interacting. If your site has high bounce rates it might be because your page is too slow.
- **Unique Users**: This metric tracks how many individual visitors come to your site. To view the Bounce Rate and Unique Users, you can use Google Analytics. It will allow you to monitor these metrics and improve engagement.

Common Performance Measurement Tools

- **Chrome DevTools**: Chrome DevTools is a built-in tool inside Google Chrome that lets you analyze and debug performance in real-time. DevTools will show loading times, CPU usage, and render

delays. It's especially useful for measuring First Contentful Paint, or FCP, is how fast a user sees the first visible content. If your website feels slow, DevTools will help you spot the bottlenecks.

- **Lighthouse**: This is an automated tool that checks performance, SEO, and accessibility.
- **WebPageTest**: This tool lets you test how your site loads from different locations and devices. This tool gives you a detailed breakdown of your site's Speed Index, Total Blocking Time, and other key performance metrics. If you want to know how real users experience your site globally, WebPageTest is the tool for that.
- **PageSpeed Insights**: This tool analyzes your website and suggests quick improvements for both mobile and desktop. It will tell you what's slowing your site down and give specific recommendations like optimizing images, removing render-blocking scripts, and reducing server response times. PageSpeed Insights is a fast and easy way to check how Google sees your site's performance.
- **Real User Monitoring**: RUM tools track actual user behavior, showing how real visitors experience your site. Popular RUM tools include Google Analytics, which tracks page load times and bounce rates, and New Relic or Datadog, which monitor real-time performance issues. If you want data from actual users, RUM tools are essential.

Working with Performance Web APIs

- **Definition**: Performance Web APIs let developers track how efficiently a webpage loads and responds directly in the code. These APIs allow you to measure page load times, track rendering and interaction delays and analyze JavaScript execution time.
- `performance.now()`: This API gives you high-precision timestamps (in milliseconds) to measure how long different parts of your site take to load.

```
const start = performance.now();
// Run some code here
const end = performance.now();

console.log(`Execution time: ${end - start}ms`);
```

- **Performance Timing API**: This API gives you a breakdown of every stage of page loading, from DNS lookup to `DOMContentLoaded`.

```
const timing = performance.timing;

const pageLoadTime = timing.loadEventEnd - timing.navigationStart;
console.log(`Page load time: ${pageLoadTime}ms`);
```

- `PerformanceObserver`: This API listens for performance events such as layout shifts, long tasks, and user interactions.

```
const observer = new PerformanceObserver((list) => {
  list.getEntries().forEach((entry) => {
    console.log(`Long task detected: ${entry.duration}ms`);
  });
});

observer.observe({ type: "longtask", buffered: true });
```

Techniques for Improving CSS Performance

- **CSS Animations**: Animating certain CSS properties, such as dimensions, position, and layout, triggers a process called "reflow", during which the browser recalculates the position and geometry of certain elements on the page. This requires a repaint, which is computationally expensive.

Therefore, it's recommended to reduce the number of CSS animations as much as possible or at least give the user an option to toggle them on or off.

Techniques for Improving JavaScript Performance

- **Code Splitting**: Splitting your JavaScript code into modules that perform critical and non-critical tasks is also helpful. This way, you'll be able to preload the critical ones as soon as possible and defer the non-critical ones to render the page as fast as possible.
- **DOM Manipulation**: Remember that DOM Manipulation refers to the process of dynamically changing the content of a page with JavaScript by interacting with the Document Object Model (DOM). Manipulating the DOM is computationally expensive. By reducing the amount of DOM manipulation in your JavaScript code will improve performance.

CSS Frameworks

- **CSS frameworks**: CSS frameworks can speed up your workflow, create a uniform visual style across a website, make your design look consistent across multiple browsers, and keep your CSS code more organized.
- **Popular CSS frameworks**: Some of the popular CSS frameworks are Tailwind CSS, Bootstrap, Materialize, and Foundation.
- **Potential disadvantages**:
  - The CSS provided by the framework might conflict with your custom CSS.
  - Your website might look similar to other websites using the same framework.
  - Large frameworks might cause performance issues.

Two Types of CSS Frameworks

- **Utility-first CSS frameworks**: These frameworks have small classes with specific purposes, like setting the margin, padding, or background color. You can assign these small classes directly to the HTML elements as needed. Tailwind CSS is categorized as a utility-first CSS framework.

Here is an example of using Tailwind CSS to style a button.

```
<button class="bg-blue-500 text-white font-bold py-2 px-4 rounded-full
hover:bg-blue-700">
  Button
</button>
```

- **Component-based CSS frameworks**: These frameworks have pre-built components with pre-defined styles that you can add to your website. The components are available in the official documentation of the CSS framework, and you can copy and paste them into your project. Bootstrap is categorized as a component-based CSS framework.

Here is an example of using Bootstrap to create a list group. Instead of applying small classes to your HTML elements, you will add the entire component, including the HTML structure.

```
<div class="card" style="width: 25rem;">
  <ul class="list-group list-group-flush">
    <li class="list-group-item">HTML</li>
    <li class="list-group-item">CSS</li>
    <li class="list-group-item">JavaScript</li>
  </ul>
</div>
```

Tailwind CSS

Tailwind is a utility-first CSS framework. Instead of writing custom CSS rules, you build your designs by combining small utility classes directly in your HTML.

Responsive Design Utilities

Tailwind uses prefixes such as `sm:`, `md:`, and `lg:` to apply styles at different screen sizes.

```
<div class="w-full md:w-1/2 lg:flex-row">
  Responsive layout
</div>
```

Flexbox Utilities

Classes like `flex`, `flex-col`, `justify-around`, and `items-center` make it easy to create flexible layouts.

```
<div class="flex flex-col md:flex-row justify-around items-center">
  <p>Column on small screens</p>
  <p>Row on medium and larger screens</p>
</div>
```

Grid Utilities

Tailwind includes utilities for CSS Grid, like `grid`, `grid-cols-1`, and `md:grid-cols-3`.

```
<div class="grid grid-cols-1 md:grid-cols-3 gap-8">
  <div class="bg-gray-100 p-4">Column 1</div>
  <div class="bg-gray-100 p-4">Column 2</div>
  <div class="bg-gray-100 p-4">Column 3</div>
</div>
```

Spacing Utilities

Utilities like `mt-8`, `mx-auto`, `p-4`, and `gap-4` help create consistent spacing without writing CSS.

```
<div class="mt-8 p-4 bg-indigo-600 text-white">
  Spaced content
</div>
```

Typography Utilities

Utilities like `uppercase`, `font-bold`, `font-semibold`, and `text-4xl` control text appearance.

You can set font sizes that adjust at breakpoints, such as `text-3xl` and `md:text-5xl`.

```
<h1 class="text-3xl md:text-5xl font-semibold text-center">
  Responsive Heading
</h1>
```

Colors and Hover States

Tailwind provides a wide color palette, such as `text-red-700`, `bg-indigo-600`, and `bg-gray-100`.

Classes like `hover:bg-pink-600` make interactive effects simple.

```
<a href="#" class="bg-pink-500 hover:bg-pink-600 text-white px-4 py-2
rounded-md">
  Hover Me
</a>
```

Borders, Rings, and Effects

- **Borders**: `border-2 border-red-300` adds borders with specified thickness and colors.
- **Rings**: `ring-1 ring-gray-300` creates outline-like effects often used for focus or cards.
- **Rounded corners and scaling**: Classes like `rounded-md`, `rounded-xl`, and `scale-105` add polish.

```
<div class="p-6 rounded-xl ring-2 ring-fuchsia-500 scale-105">
  Highlighted card
</div>
```

Gradients

Tailwind supports gradient utilities like `bg-gradient-to-r from-fuchsia-500 to-indigo-600`.

```
<div class="p-4 text-white bg-gradient-to-r from-fuchsia-500 to-indigo-600">
  Gradient background
</div>
```

CSS Preprocessors

- **CSS preprocessor**: A CSS preprocessor is a tool that extends standard CSS. It compiles the code with extended syntax into a native CSS file. It can be helpful for writing cleaner, reusable, less repetitive, and scalable CSS for complex projects.
- **Features**: Some of the features that can be provided by CSS preprocessors are variables, mixins, nesting, and selector inheritance.
- **Popular CSS preprocessors**: Some of the popular CSS preprocessors are Sass, Less, and Stylus.
- **Potential disadvantages**:
  - Compiling the CSS rules into standard CSS might cause overhead.
  - The compiled code may be difficult to debug.

Sass

- **Sass**: It is one of the most popular CSS preprocessors. Sass stands for "Syntactically Awesome Style Sheets."
- **Features supported by Sass**: Sass supports features like variables, nested CSS rules, modules, mixins, inheritance, and operators for basic mathematical operations

Two Syntaxes Supported by Sass

- **SCSS syntax**: The SCSS (Sassy CSS) expands the basic syntax of CSS. It is the most widely used syntax for Sass. SCSS files have an `.scss` extension.

Here is an example of defining and using a variable in SCSS.

```
$primary-color: #3498eb;

header {
  background-color: $primary-color;
}
```

- **Indented syntax**: The indented syntax was Sass's original syntax and is also known as the "Sass syntax."

Here is an example of defining and using a variable in the indented syntax.

```
$primary-color: #3498eb

header
  background-color: $primary-color
```

Mixins

- **Mixins**: Mixins allow you to group multiple CSS properties and their values under the name and reuse that block of CSS code throughout your stylesheet.

Here is an example of defining a mixin in SCSS syntax. In this case, the mixin is called `center-flex`. It has three CSS properties to center elements using flexbox.

```
@mixin center-flex {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Here is an example of using the mixin you defined.

```
section {
  @include center-flex;
  height: 500px;
  background-color: #3289a8;
}
```

Manual and Automated Testing

- **Manual Testing**: In manual testing, a tester will manually go through each part of the application and test out different features to make sure it works correctly. If any bugs are uncovered in the testing process, the tester will report those bugs back to the software team so they can be fixed.
- **Automated Testing**: In automated testing, you can automate your tests by writing a separate program that checks whether your application behaves as expected.

Unit Testing

- **Unit Testing**: In unit testing, you test each function to ensure that everything is working as expected. Unit tests can also serve as a form of documentation for your application because they are meant to represent the expected behavior for your code.
- **Single Responsibility Principle**: The single responsibility principle recommends keeping each function small and responsible for one thing.
- **Common JavaScript Testing Frameworks**: Some common testing frameworks include Jest, Mocha, and Vitest. Jest is a popular testing framework for unit tests.

Here is an example of unit tests using Jest.

First, you can create a function that is responsible for returning a newly formatted string:

```
```

```
export function getFormattedWord(str) {
  if (!str) return '';
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

In a separate `getFormattedWord.test.js` file, you can write some tests to verify that the function is working as expected. The `getFormattedWord.test.js` file will look like this:

```
import { getFormattedWord } from "./getFormattedWord.js";
import { test, expect } from "jest";

test('capitalizes the first letter of a word', () => {
  expect(getFormattedWord('hello')).toBe('Hello');
});
```

- **`expect` Function**: The `expect` function is used to test a value.
- **Matcher**: Matcher is a function that checks whether the value behaves as expected. In the above example, the matcher is `toBe()`. Jest has a variety of matchers.

To use Jest, you first need to install the `jest` package by using `npm i jest`. You will also need to add a script to your `package.json` file like this:

```
"scripts": {
  "test": "jest"
},
```

Then, you can run the command `npm run test` to run your tests.

Software Development Lifecycle

- **Different Stages of Software Development Lifecycle**:
  - **Planning Stage**: The development team collects requirements for the proposed work from the stakeholders.
  - **Design Stage**: The software team breaks down the requirements and decides on the best approaches for solutions.
  - **Implementation Stage**: The software team breaks down the requirements into manageable tasks and implements them.
  - **Testing Stage**: This involves manual and automated testing for the new work. Sometimes, the team tests out the application throughout the entire development stage to catch and fix any issues that come up.
  - **Deployment Stage**: The team deploys the new changes to a build or testing environment.
  - **Maintenance Stage**: This involves fixing any issues that arise from customers in the production application.
- **Different Models of the Software Development Lifecycle**:
  - **Waterfall Model**: The Waterfall model is where each phase of the lifecycle needs to be completed before the next phase can begin.
  - **Agile Model**: The Agile model focuses on iterative development by breaking down work into sprints.

BDD and TDD

- **TDD**: Test-driven development is a methodology that emphasizes writing tests first. Writing tests before building out features provides real-time feedback to developers during the development

process.
- **BDD**: Behavior-driven development is the approach of aligning a series of tests with business goals. The test scenarios in BDD should be written in a language that can be understood by both technical and non-technical individuals. An example of such syntax is Gherkin.
- **BDD Testing Frameworks**: Examples of BDD testing frameworks include Cucumber, JBehave, and SpecFlow.

Assertions in Unit Testing

- **Assertion**: Assertions are used to test that the code is behaving as expected.
- **Assertion Libraries**: Chai is a commonly used assertion library. Other common JavaScript assertion libraries are `should.js` and `expect.js`.

Here is an example of an assertion using Chai that checks that the return value from the `addThreeAndFour` function is equal to the number 7:

```
assert.equal(addThreeAndFour(), 7);
```

- **Best Practices**: Regardless of which assertion library you use, you should write clear assert and failure messages that will help you understand which tests are failing and why.

Mocking, Faking, and Stubbing

- **Mocking**: Mocking is the process of replacing real data with false data that simulates the behavior of real components. For example, you could mock the API response in testing instead of making continuous API calls to fetch the data.
- **Stubbing**: Stubs are objects that return pre-defined responses or dummy data for an expected behavior in an application. For example, you can stub the behavior for a database connection in your tests without having to rely on an actual database connection.
- **Faking**: Fakes are simplified versions of real components without the complexity or side effects. For example, you can fake a database by storing the data in memory instead of interacting with the real database. This will allow you to mimic database operations in memory, which will be much faster than dealing with the real database.

Functional Testing

- **Functional Testing**: Functional testing checks if the features and functions of the application work as expected. The goal of functional testing is to test the system as a whole against multiple scenarios.
- **Non-Functional Testing**: Non-functional testing focuses on things like performance and reliability.
- **Smoke testing**: Smoke testing is a preliminary check on the system for basic or critical issues before beginning more extensive testing.

End-to-End Testing

- **End-to-End Testing**: End-to-end testing, or E2E, tests real-world scenarios from the user's perspective. End-to-end tests help ensure that your application behaves correctly and is predictable for users. However, it is time-consuming to set up, design, and maintain.
- **End-to-End Testing Frameworks**: Playwright is a popular end-to-end testing framework developed by Microsoft. Other examples of end-to-end testing tools include Cypress, Selenium, and Puppeteer.

Here is an example of E2E tests from the freeCodeCamp codebase using Playwright.
The `beforeEach` hook will run before each of the tests. The tests check that the donor has a supporter link in the menu bar, as well as a special stylized border around their avatar:

```
test.beforeEach(async ({ page }) => {
  execSync("node ./tools/scripts/seed/seed-demo-user --set-true isDonating");
```

```
  await page.goto("/donate");
});

...

test("The menu should have a supporters link", async ({ page }) => {
  const menuButton = page.getByTestId("header-menu-button");
  const menu = page.getByTestId("header-menu");

  await expect(menuButton).toBeVisible();
  await menuButton.click();

  await expect(menu).toBeVisible();

  await expect(page.getByRole("link", { name: "Supporters" })).toBeVisible();
});

test("The Avatar should have a special border for donors", async ({ page })
=> {
  const container = page.locator(".avatar-container");
  await expect(container).toHaveClass("avatar-container gold-border");
});
```

Usability Testing

- **Usability Testing**: Usability testing is when you have real users interacting with the application to discover if there are any design, user experience, or functionality issues in the app. Usability testing focuses on the intuitiveness of the application by users.
- **Four Common Types of Usability Testing**:
  - **Explorative**: Explorative usability testing involves users interacting with the different features of the application to better understand how they work.
  - **Comparative**: Comparative testing is where you compare your application's user experience with similar applications in the marketplace.
  - **Assessment**: Assessment testing is where you study how intuitive the application is to use.
  - **Validation**: Validation testing is where you identify any major issues that will prevent the user from using the application effectively.
- **Usability Testing Tools**: Examples of tools for usability testing include Loop11, Maze, Userbrain, UserTesting, and UXTweak.

Compatibility Testing

- **Compatibility Testing**: The goal of compatibility testing is to ensure that your application works in different computing environments.
- **Different Types of Compatibility Testing**:
  - **Backwards Compatibility**: Backwards compatibility refers to when the software is compatible with earlier versions.
  - **Forwards Compatibility**: Forwards compatibility refers to when the software and systems will work with future versions.
  - **Hardware Compatibility**: Hardware compatibility is the software's ability to work properly in different hardware configurations.
  - **Operating Systems Compatibility**: Operating systems compatibility is the software's ability to work on different operating systems, such as macOS, Windows, and Linux distributions like Ubuntu and Fedora.
  - **Network Compatibility**: Network compatibility means the software can work in different network conditions, such as different network speeds, protocols, security settings, etc.
  - **Browser Compatibility**: Browser compatibility means the web application can work consistently across different browsers, such as Google Chrome, Safari, Firefox, etc.
  - **Mobile Compatibility**: It is important to ensure that your software applications work on a variety of Android and iOS devices, including phones and tablets.

Performance Testing

- **Performance Testing**: In performance testing, you test an application's speed, responsiveness, scalability, and stability under different workloads. The goal is to resolve any type of performance bottleneck.
- **Different Types of Performance Testing**:
  - **Load Testing**: Load testing determines how a system behaves during normal and peak load times.
  - **Stress Testing**: Stress testing is where you test your application in extreme loads and see how well your system responds to the higher load.
  - **Soak Testing (Endurance Testing)**: Soak testing or endurance testing is a type of load testing where you test the system with a higher load for an extended period of time.
  - **Spike Testing**: Spike testing is where you dramatically increase and decrease the loads and analyze the system's reactions to the changes.
  - **Breakpoint Testing (Capacity Testing)**: Breakpoint testing or capacity testing is where you slowly increment the load over time to the point where the system starts to fail or degrade.

Security Testing

- **Security Testing**: Security testing helps identify vulnerabilities and weaknesses.
- **Security Principles**:
  - **Confidentiality**: This protects against the release of sensitive information to other recipients that aren't the intended recipient.
  - **Integrity**: This involves preventing malicious users from modifying user information.
  - **Authentication**: This involves verifying the user's identity to ensure that they are allowed to use that system.
  - **Authorization**: This is the process of determining what actions authenticated users are allowed to perform or which parts of the system they are permitted to access.
  - **Availability**: This ensures that information and services are available to authorized users when they need it.
  - **Non-Repudiation**: This ensures that both the sender and recipient have proof of delivery and verification of the sender's identity. It protects against the sender denying having sent the information.
- **Common Security Threats**:
  - **Cross-Site Scripting (XSS)**: XSS attacks happen when an attacker injects malicious scripts into a web page and then executes them in the context of the victim's browser.
  - **SQL Injection**: SQL injection allows malicious users to inject malicious code into a database.
  - **Denial-of-Service (DoS) Attack**: DoS attack is when malicious users flood a website with a high number of requests or traffic, causing the server to slow down and possibly crash, making the site unavailable to users.
- **Categories of Security Testing Tools**:
  - **Static Application Security Testing**: These tools evaluate the source code for an application to identify security vulnerabilities.
  - **Dynamic Application Security Testing**: These tools interface with the application's frontend to uncover potential security weaknesses. DAST tools do not have access to the source code.
- **Penetration Testing (pentest)**: Penetration testing is a type of security testing that involves creating simulated cyberattacks on the application to identify any vulnerabilities in the system.

A/B Testing

- **A/B Testing**: A/B testing involves comparing two versions of a page or application and studying which version performs better. It is also known as bucket or split testing. A/B testing allows you to make more data-driven decisions and continually improve the user experience.
- **Tools for A/B Testing**: Examples of tools to use for A/B testing include GrowthBook and LaunchDarkly.

Alpha and Beta Testing

Once the initial development and software testing are complete, it is important to have the application tested by testers and real users. This is where alpha and beta testing come in.

- **Alpha Testing**: Alpha testing is done by a select group of testers who go through the application to ensure there are no bugs before it is released into the marketplace. Alpha testing is part of acceptance testing and utilizes both white and black box testing techniques.
- **Beta Testing**: Beta testing is where the application is made available to real users. Users can interact with the application and provide feedback. Beta testing is also a form of user acceptance testing.
- **Acceptance Testing**: Acceptance testing ensures that the software application meets the business requirements and the needs of users before its release.
- **Black Box Testing**: Black box testing only focuses on the expected behavior of the application.
- **White Box Testing**: White box testing involves the tester knowing the internal components and performing tests on them.

Regression Testing

- **Regression**: Regression refers to situations where new changes unintentionally break existing functionality.
- **Regression Testing**: Regression testing helps catch regression issues. In regression testing, you re-run functional tests against parts of your application to ensure that everything still works as expected.
- **Tools for Regression Testing**: Tools that you can use to perform regression testing include Puppeteer, Playwright, Selenium, and Cypress.
- **Techniques for Regression Testing**:
    - **Unit Regression Testing**: This is where you have a list of items that need to be tested each time major changes or fixes are implemented into the app.
    - **Partial Regression Testing**: This involves targeted approaches to ensure that new changes have not broken specific aspects of the application.
    - **Complete Regression Testing**: This runs tests against all the functionalities in the codebase. This is the most time-consuming and detailed option.
- **Retesting**: Retesting is used to check for known issues and ensure that they have been resolved. In contrast, regression testing searches for unknown issues that might have occurred through recent changes in the codebase.

What is TypeScript

- **JavaScript**: JavaScript is a dynamically-typed language. This means that variables can receive any values at run time. The challenge of a dynamically-typed language is that the lack of type safety can introduce errors.

For example, even if your JavaScript function expects an array, you can still call it with a number:

```
const getRandomValue = (array) => {
  return array[Math.floor(Math.random() * array.length)];
}

console.log(getRandomValue(10));
```

The `console` output for the example above will be `undefined`.

- **TypeScript**: TypeScript extends the JavaScript language to include static typing, which helps catch errors caused by type mismatches before you run your code.

For example, you can define a type for the `array` parameter as follows:

```
const getRandomValue = (array: string[]) => {
  return array[Math.floor(Math.random() * array.length)];
```

```
}
```

This type definition tells TypeScript that the `array` parameter must be an array of strings. Then, when you call `getRandomValue` and pass it a number, you get an error called a compiler error.

- **Compiler**: You first need to compile TypeScript code into regular JavaScript. When you run the compiler, TypeScript will evaluate your code and throw an error for any issues where the types don't match.

Data Types in TypeScript

- **Primitive Data Types in TypeScript**: For the primitive data types `string`, `null`, `undefined`, `number`, `boolean`, and `bigint`, TypeScript offers corresponding type keywords.

```
let str: string = "Naomi";
let num: number = 42;
let bool: boolean = true;
let nope: null = null;
let nada: undefined = undefined;
```

- **Array**: You can define an array of specific type with two different syntaxes.

```
const arrOne: string[] = ["Naomi"];
const arrTwo: Array<string> = ["Camperchan"];
```

- **Objects**: You can define the exact structure of an object.

```
const obj: { a: string, b: number } = { a: "Naomi", b: 42 };
```

If you want an object with any keys, but where all values must be strings, there are two ways to define it:

```
const objOne: Record<string, string> = {};
const objTwo: { [key: string]: string } = {};
```

- **Other Helpful Types in TypeScript**:
  - **`any`**: `any` indicates that a value can have any type. It tells the compiler to stop caring about the type of that variable.
  - **`unknown`**: `unknown` tells TypeScript that you *do* care about the type of the value, but you don't actually know what it is. `unknown` is generally preferred over `any`.
  - **`void`**: This is a special type that you'll typically only use when defining functions. Functions that don't have a return value use a return type of `void`.
  - **`never`**: It represents a type that will never exist.

- **`type` Keyword**: This keyword is like `const`, but instead of declaring a variable, you can declare a type.

It is useful for declaring custom types such as union types or types that include only specific values:

```
type stringOrNumber = string | number;
type bot = "camperchan" | "camperbot" | "naomi";
```

- `interface`: Interfaces are like classes for types. They can implement or extend other interfaces, are specifically object types, and are generally preferred unless you need a specific feature offered by a `type` declaration.

```
interface wowie {
  zowie: boolean;
  method: () => void;
}
```

- **Defining Return Type**: You can also define the *return type* of the function.

The example below defines the return value as a string. If you try to return anything else, TypeScript will give a compiler error.

```
const getRandomValue = (array: string[]): string => {
  return array[Math.floor(Math.random() * array.length)];
}
```

Generics

- **Defining Generic Type**: You can define a generic type and refer to it in your function. It can be thought of as a special parameter you provide to a function to control the behavior of the function's type definition.

Here is an example of defining a generic type for a function:

```
const getRandomValue = <T>(array: T[]): T => {
  return array[Math.floor(Math.random() * array.length)];
}
const val = getRandomValue([1, 2, 4])
```

The `<T>` syntax tells TypeScript that you are defining a generic type `T` for the function. `T` is a common name for generic types, but you can use any name.

Then, you tell TypeScript that the `array` parameter is an array of values matching the generic type, and that the returned value is a single element of that same type.

- **Specifying Type Argument in Function Call**: You can pass a type argument to a function call using angle brackets between the function name and its parameters.

Here is an example of passing a type argument to a function call:

```
const email = document.querySelector<HTMLInputElement>("#email");
console.log(email.value);
```

This tells TypeScript that the element you expect to find will be an input element.

Type Narrowing

- **Narrowing by Truthiness**: In the example below, you get a compiler error trying to access the `value` property of `email` because `email` *might* be `null`.

```
const email = document.querySelector<HTMLInputElement>("#email");
console.log(email.value);
```

You can use a conditional statement to confirm `email` is *truthy* before accessing the property:

```
const email = document.querySelector<HTMLInputElement>("#email");
if (email) {
  console.log(email.value);
}
```

Truthiness checks can also work in the reverse direction:

```
const email = document.querySelector<HTMLInputElement>("#email");
if (!email) {
  throw new ReferenceError("Could not find email element!")
}
console.log(email.value);
```

Throwing an error ends the logical execution of this code, which means when you reach the `console.log()` call, TypeScript knows `email` *cannot* be `null`.

- **Optional Chaining**: Optional chaining `?.` is also a form of type narrowing, under the same premise that the property access can't happen if the `email` value is `null`.

```
const email = document.querySelector<HTMLInputElement>("#email");
console.log(email?.value);
```

- **`typeof` Operator**: You can use a conditional to check the type of the variable using the `typeof` operator.

```
const myVal = Math.random() > 0.5 ? 222 : "222";
if (typeof myVal === "number") {
  console.log(myVal / 10);
}
```

- **`instanceof` Keyword**: If the object comes from a class, you can use the `instanceof` keyword to narrow the type.

```
const email = document.querySelector("#email");

if (email instanceof HTMLInputElement) {
    console.log(email.value);
}
```

- **Casting**: When TypeScript cannot automatically determine the type of a value, such as the result from `request.json()` method in the example below, you'll get a compiler error. One way to resolve this is by casting the type, but doing so weakens TypeScript's ability to catch potential errors.

```
interface User {
    name: string;
    age: number;
}

const printAge = (user: User) =>
```

```
  console.log(`${user.name} is ${user.age} years old!`)

const request = await fetch("url")
const myUser = await request.json() as User;
printAge(myUser);
```

- **Type Guard**: Instead of casting the type, you can write a type guard:

```
interface User {
    name: string;
    age: number;
}

const isValidUser = (user: unknown): user is User => {
  return !!user &&
    typeof user === "object" &&
    "name" in user &&
    "age" in user;
}
```

The `user is User` syntax indicates that your function returns a boolean value, which when true means the `user` value satisfies the `User` interface.

`tsconfig` File

- **`tsconfig.json`**: TypeScript's compiler settings live in a `tsconfig.json` file in the root directory of your project.

```
{
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./prod",
    "lib": ["ES2023"],
    "target": "ES2023",
    "module": "ES2022",
    "moduleResolution": "Node",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "strict": true
  },
  "exclude": ["test/"]
}
```

Here are descriptions of the compiler options used in the example above:

- **`compilerOptions`**: The `compilerOptions` property is where you control how the TypeScript compiler behaves.
- **`rootDir` and `outDir`**: The `rootDir` and `outDir` tell TypeScript which directory holds your source files, and which directory should contain the transpiled JavaScript code.
- **`lib`**: The `lib` property determines which type definitions the compiler uses, and allows you to include support for specific ES releases, the DOM, and more.
- **`module` and `moduleResolution`**: The `module` and `moduleResolution` work in tandem to manage how your package uses modules - either CommonJS or ECMAScript.
- **`esModuleInterop`**: The `esModuleInterop` allows for smoother interoperability between CommonJS and ES modules by automatically creating namespace objects for imports.
- **`skipLibCheck`**: The `skipLibCheck` option skips validating `.d.ts` files that aren't referenced by imports in your code.

- **`strict`**: The `strict` flag enables several checks, such as ensuring proper handling of nullable types and warning when TypeScript falls back to `any`.
- **`exclude`**: The top-level `exclude` property tells the compiler to ignore these TypeScript files during compilation.

--assignment--

Review the Front End Libraries topics and concepts.

Git Review

Introduction to Version Control

- **Definition**: A version control system allows you to track and manage changes in your project. Examples of version control systems used in software are Git, SVN, or Mercurial.

Cloud-Based Version Control Providers

- **List of Cloud-Based Version Control Providers**: GitHub and GitLab are popular examples of cloud-based version control providers that allow software teams to collaborate and manage repositories.

Installing and Setting up Git

- **Installing Git**: To check if Git is already installed on your machine you can run the following command in the terminal:

```
git --version
```

If you see a version number, that means Git is installed. If not, then you will need to install it.

For Linux systems, Git often comes preinstalled with most distros. If you do not have Git pre-installed, you should be able to install it with your package manager commands such as `sudo apt-get install git` or `sudo pacman -S git`.

For Mac users, you can install Git via Homebrew with `brew install git`, or you can download the executable installer from Git's website.

For Windows, you can download the executable installer from Git's website. Or, if you have set up Chocolatey, you can run `choco install git.install` in PowerShell. Note that on Windows, you may also want to download Git Bash so you have a Unix-like shell environment available.

To make sure the installation worked, run the `git --version` command again in the terminal.

- **Git Configurations**: `git config` is used to set configuration variables that are responsible for how Git operates on your machine. To view your current setting variables and where they are stored on your system, you can run the following command:

```
git config --list --show-origin
```

Right now you should be seeing only system-level configuration settings if you just installed Git for the first time.

To set your user name, you can run the following command:

```
```

```
git config --global user.name "Jane Doe"
```

The `--global` flag is used here to set the user name for all projects on your system that use Git. If you need to override the user name for a particular project, then you can run the command in that particular project directory without the `--global` flag.

To set the user email address, you can run the following command:

```
git config --global user.email janedoe@example.com
```

Another configuration you can set is the preferred editor you want Git to use. Here is an example of how to set your preferred editor to Emacs:

```
git config --global core.editor emacs
```

If you choose not to set a preferred editor, then Git will default to your system's default editor.

Open vs. Closed Source Software

- **Definition**: "Open-source" means people can see the code you publish, propose changes, report issues, and even run a modified version. "Closed-source" means the only people who can see and interact with the project are the people you explicitly authorize.

GitHub

- **Definition**: GitHub is a cloud-based solution that offers storage of version-controlled projects in something called "repositories", and enables collaboration features to use with those projects.
- **GitHub CLI**: This tool is used to do GitHub-specific tasks without leaving the command line. If you do not have it installed, you can get instructions to do so from GitHub's documentation - but you should have it available in your system's package manager.
- **GitHub Pages**: GitHub Pages is an option for deploying static sites, or applications that do not require a back-end server to handle logic. That is, applications that run entirely client-side, or in the user's browser, can be fully deployed on this platform.
- **GitHub Actions**: GitHub Actions is a feature that lets you automate workflows directly in your GitHub repository including building, testing, and deploying your code.

Common Git Commands

- **`git init`**: This will initialize an empty Git repository so Git can begin tracking changes for this project. When you initialize an empty Git repository to a project, a new `.git` hidden directory will be added. This `.git` directory contains important information for Git to manage your project.
- **`git status`**: This command is used to show the current state of your working directory - you will be using this command a lot in your workflow.
- **`git add`**: This command is used to stage your changes. Anything in the staging area will be added for the next commit. If you want to stage all unstaged changes, then you can use `git add .` The period (`.`) is an alias for the current directory you are in.
- **`git commit`**: This command is used to commit your changes. A commit is a snapshot of your project state at that given time. If you run `git commit`, it will open up your preferred editor you set in the Git configuration. Once the editor is open, you can provide a detailed message of your changes. You can also choose to provide a shorter message by using the `git commit -m` command like this:

```
git commit -m "short message goes here"
```

- `git log`: This will list all prior commits with helpful information like the author, date of commit, commit message and commit hash. The commit hash is a long string which serves as a unique identifier for a commit.
- `git remote add`: This command is used to setup the remote connection to your remote repo.
- `git push`: This command is used to push up your changes to a remote repository.
- `git pull`: This command is used to pull down the latest changes from your remote repository into your local repository.
- `git clone`: This command will clone a repository. This means you will have a copy of the repository. This copy includes the repository history, all files/folders and commits on your local device.
- `git remote -v`: This command will show the list of remote repositories associated with your local Git repository.
- `git branch`: This command will list all of your local branches.
- `git fetch upstream`: This command tells Git to go get the latest changes that are on your upstream remote (which is the original repo).
- `git merge upstream/main`: This command tells Git to merge the latest changes from the `main` branch in the upstream remote into your current branch.
- `git reset`: This command allows you to reset the current state of a branch. Passing the `--hard` flag tells Git to force the local files to match the branch state. This ensures that you have a clean slate to work from.
- `git rebase`: A rebase in Git is a way to move or combine a sequence of commits from one branch onto another.

Working with Branches

- **Definition**: A branch in Git is a separate workspace where you can make changes. The `main` branch will often represent the primary or production branch in a real world application. Developer teams will create multiple branches for new features and bug fixes and then merge those changes back into the `main` branch.
- **Creating a New Branch**: To create a new branch you can run the following command:

```
git branch feature
```

To checkout that branch, you can run the following command:

```
git checkout feature
```

Most developers will use the shorthand command for creating and checking out a branch which is the following:

```
git checkout -b new-branch-name
```

A newer and alternative command would be the `git switch` command. Here is an example for creating and switching to a new branch:

```
git switch -c new-branch-name
```

- **Branching Strategies**: Your `main` branch is your default branch and usually is pretty stable. So it is best to branch off from there to create new branches for items like bug fixes, new features, or other miscellaneous work.
- **Merge Conflicts**: This happens when Git tries to automatically merge changes from different branches but can't decide which changes to keep. This usually happens when there are conflicting

changes for the same portion of the file.

Five States for a Git Tracked File

- **"Untracked"**: This means that the file is new to the repository, and Git has not "seen" it before.
- **"Modified"**: This file existed in the previous commit, and has changes that have not been committed.
- **"Ignored"**: You likely won't see ignored files in Git, but your IDE might have an indicator for them. Ignored files are excluded from Git operations, typically because they are included in the `.gitignore` file.
- **"Deleted"**: A deleted file is the opposite of an untracked file - it's a file that previously existed, and has been removed.
- **"Renamed"**: A renamed file is a file where the contents are unchanged, but the name or location of the file was modified. In some cases, a file can be considered renamed even if it has a small amount of changes.

`.gitignore` Files

- **Definition**: The `.gitignore` file is a special type of file related to Git operations. The name suggests that this file is used to tell Git to ignore things, and that's the common use case. But what it actually does is it tells Git to stop tracking a file.

Working with Repositories

- **Definition**: A repository is like a container for a project - if you are working on an app, you would keep the files for that app together in a repository. Repositories can be local on your computer, or remote on a service like GitHub.
- **Public vs. Private Repositories**: A public repository can be viewed and downloaded by anyone. A private repository can only be accessed by you, and anyone you grant explicit access to.
- **Creating Repositories on GitHub**: To create a new repository on GitHub, you can click on the `"New Repository"` button and walk through the GitHub UI of setting up a new repository.
- **Pushing Local Repositories to GitHub**: If you have a local project on your computer, you can push up that repository to GitHub. Here is a step-by-step overview of the process:

1. Initialize an empty git repository in the project directory (`git init`).
2. Make changes to your project.
3. Run the `git status` command to see all changes made that are being tracked by git.
4. Stage your changes (`git add`).
5. Commit your changes (`git commit`).
6. Setup the remote connection (`git remote add`).
7. Push your changes to GitHub (`git push`).

Pull Requests

- **Pull Requests**: A pull request is a request to pull changes in from your branch into the target branch. Pull requests are the flow you use when you want to contribute code changes to a project. This approach allows the maintainers of the project to review your changes. They can leave comments, ask questions, and suggest tweaks. Then once the review process is complete, it can be approved and merged into the main branch.

Contributing to Other Repositories

- **Process**: There are thousands of projects that you can contribute to. Here is the basic process on how to contribute to another repository:

1. Read the contributing documentation
2. Find an available issue to work on
3. Fork the repository
4. Clone your forked copy of the repository

5. Create a new branch
6. Make the changes according to the issue
7. Create a PR (Pull Request)
8. Wait for a review for that PR

Working with SSH and GPG Keys

- **GPG Keys**: GPG, or Gnu Privacy Guard, keys are typically used to sign files or commits. Someone can then use your public GPG key to verify that the file signature is from your key and that the contents of the file have not been modified or tampered with.

To generate a GPG key, you'll need to run:

```
gpg --full-generate-key
```

- **SSH Keys**: SSH, or Secure SHell, keys are typically used to authenticate a remote connection to a server - via the `ssh` utility. You can also use an SSH key to sign commits.

For an SSH key, you'll run:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

`ed25519` is a modern public-key signature algorithm.

- **Signing Commits with GPG Keys**: In order to sign your commits with your GPG key, you'll need to upload your public key, not the private key, to your GitHub account. To list your public keys, you will need to run the following:

```
gpg --list-secret-keys --keyid-format=long
```

Then, to get the public key, use:

```
gpg --armor --export "<key id>"
```

Then, take the short ID you got from listing the keys and run this command to set it as your git signing key:

```
git config --global user.signingkey <your_gpg_key_id>
```

Then, you can pass the `-S` flag to your `git commit` command to sign a specific commit - you'll need to provide your passphrase. Alternatively, if you want to sign every commit automatically, you can set the autosign config to `true`:

```
git config --global commit.gpgsign true
```

- **Signing Commits with SSH Keys**: To sign with an SSH key, which is a relatively new feature on GitHub, you'll need to start by uploading the key to your GitHub account. Then you'll need to set the signing mode for git to use SSH:

```
git config --global gpg.format ssh
```

Then, to set the signing key, you'll pass the file path instead of an ID:

```
git config --global user.signingkey <path_to_your_ssh_keys>
```

--assignment--

Review Git topics and concepts.

Graphs and Trees Review

Graphs Overview

A graph is a set of nodes (vertices) connected by edges (connections). Each node can connect to multiple other nodes, forming a network. The different types of graphs include:

- Directed: edges have a direction (from one node to another), often represented with straight lines and arrows.
- Undirected: edges have no direction, represented with simple lines.
- Vertex: each node is associated to a label or identifier.
- Cyclic: contains cycles (a path that starts and ends at the same node).
- Acyclic (DAG): does not contain cycles.
- Edge labeled: each edge has a label usually drawn next to corresponding edge.
- Weighted: edges have weights (values) associated with them, that can be used to perform arithmetic operations.
- Disconnected: contains two or more nodes that are not connected by any edges.

Graphs are used in various applications such as maps, networks, recommendation systems, dependency resolution.

Graph Traversals

This involves visiting all the nodes in a graph. The two main algorithms are:

- **Breadth-First Search (BFS)**
    - Uses a queue.
    - Explores level by level.
    - Finds shortest path in unweighted graphs.
- **Depth-First Search (DFS)**
    - Uses a stack (or recursion).
    - Explores a branch fully before backtracking.
    - Useful for cycle detection and path finding.

Graph Representations

Graphs can be represented in two main ways:

- **Adjacency List**
    - Each node has a list of its neighbors.
    - Space efficient for sparse graphs.
    - Easy to iterate over neighbors.
- **Adjacency Matrix**

    - A 2D array where rows and columns represent nodes.
    - Space intensive for large graphs.
    - Fast to check if an edge exists between two nodes.

Trees

A tree is a special type of graph that is acyclic and connected. Key properties include:

- They have no loops or cycles (paths where the start and end nodes are the same).
- They must be connected (every node can be reached from every other node).

Common types of trees

The most common types of trees are:

- Binary Trees
    - Each node has at most two children, a left and a right child.
- Binary Search Trees (BST)
    - A binary tree in which every left child is less than its parent, and every right child is greater than its parent.

Tries

Also known as prefix trees, they are used to store sets of strings, where each node represents a character.

Shared prefixes are stored only once, making them efficient for tasks like autocomplete and spell checking.

Search and insertion operations have a time complexity of O(L), where L is the length of the string.

Priority Queues

A priority queue is an abstract data type where each element has a priority.

Queues and stacks consider only the order of insertion, while priority queues consider the priority of elements.

Standard queues follow FIFO (First In First Out) and stacks follow LIFO (Last In First Out). However, in a priority queue, elements with higher priority are served before those with lower priority, regardless of their insertion order.

Heaps

It's a specialized tree-based data structure with a very specific property called the heap property.

The heap property determines the relationship between parent and child nodes. There are two types of heaps:

- Max-Heap
    - The value of each parent node is greater than or equal to the values of its children.
    - The largest element is at the root.
- Min-Heap
    - The value of each parent node is less than or equal to the values of its children.
    - The smallest element is at the root.

Python heap module example

```
```

```
import heapq

# Create empty heap
my_heap = []

# Insert elements
heapq.heappush(my_heap, 9)
heapq.heappush(my_heap, 3)
heapq.heappush(my_heap, 5)

# Remove smallest element
print(heapq.heappop(my_heap))  # 3

# Push + Pop in one step
print(heapq.heappushpop(my_heap, 2)) # 2

# Transform list into heap
nums = [5, 7, 3, 1]
heapq.heapify(nums)
```

Using Priorities

```
my_heap = []
heapq.heappush(my_heap, (3, "A"))
heapq.heappush(my_heap, (2, "B"))
heapq.heappush(my_heap, (1, "C"))

# Removes lowest number = highest priority
print(heapq.heappop(my_heap))  # (1, "C")
```

--assignment--

Review the Graphs and Trees topics and concepts.

HTML Review

Review the concepts below to prepare for the upcoming prep exam.

HTML Basics

- **Role of HTML**: HTML (Hypertext Markup Language) is the foundation of web structure, defining the elements of a webpage.
- **HTML Elements**: Used to represent content on the page. Most of them are made by an opening and a closing tag (e.g., `<h1></h1>`, `<p></p>`).
- **HTML Structure**: HTML consists of a `head` and `body`, where metadata, styles, and content are structured.
- **Common HTML Elements**: Headings (`<h1>` - `<h6>`), paragraphs (`<p>`), and div containers (`<div>`).
- **`div` elements**: The `div` element is a generic HTML element that does not hold any semantic meaning. It is used as a generic container to hold other HTML elements.
- **Void Elements**: Do not have a closing tag (e.g., `<img>`).
- **Attributes**: Adding metadata and behavior to elements.

Identifiers and Grouping

- **IDs**: Unique element identifiers.

- **Classes**: Grouping elements for styling and behavior.

Special Characters and Linking

- **HTML entities**: Using special characters like `&` and `<`.
- **`link` element**: Linking to external stylesheets.
- **`script` element**: Embedding external JavaScript files.

Boilerplate and Encoding

- **HTML boilerplate**: Basic structure of a webpage, which includes the `DOCTYPE`, `html`, `head`, and `body` elements. It should be used as the starting point for an HTML document.
- **UTF-8 character encoding**: Ensuring universal character display.

SEO and Social Sharing

- **Meta tags (`description`)**: Providing a short description for the web page and impacting SEO.
- **Open Graph tags**: Enhancing social media sharing.

Media Elements and Optimization

- **Replaced elements**: Embedded content (e.g., images, iframes).
- **Optimizing media**: Techniques to improve media performance.
- **Image formats and licenses**: Understanding usage rights and types.
- **SVGs**: Scalable vector graphics for sharp visuals.

Multimedia Integration

- **HTML audio and video elements**: Embedding multimedia.
- **Embedding with `<iframe>`**: Integrating external video content.

Paths and Link Behavior

- **Target attribute types**: Controlling link behavior.
- **Absolute vs. relative paths**: Navigating directories.
- **Path syntax**: Understanding `/`, `./`, `../` for file navigation.
- **Link states**: Managing different link interactions (hover, active).

Importance of Semantic HTML

- **Structural hierarchy for heading elements**: It is important to use the correct heading element to maintain the structural hierarchy of the content. The `h1` element is the highest level of heading and the `h6` element is the lowest level of heading.
- **Presentational HTML elements**: Elements that define the appearance of content. Ex. the deprecated `center`, `big`, and `font` elements.
- **Semantic HTML elements**: These elements provide meaning to the structure of the content. Examples include:
  - `<header>`: Represents introductory content.
  - `<nav>`: Contains navigation links.
  - `<article>`: Represents self-contained content.
  - `<aside>`: Used for sidebars or related content.
  - `<section>`: Groups related content within a document.
  - `<footer>`: Defines the footer for a section or document.

Semantic HTML Elements

- **Emphasis (`em`) element**: Marks text that has stress emphasis.
- **Idiomatic Text (`i`) element**: Used for highlighting alternative voice or mood, idiomatic terms from another language, technical terms, and thoughts.
- **Strong Importance (`strong`) element**: Marks text that has strong importance.

- **Bring Attention To (`b`) element**: Used to bring attention to text that is not important for the meaning of the content.
- **Description List (`dl`) element**: Used to represent a list of term-description groupings.
- **Description Term (`dt`) element**: Used to represent the term being defined.
- **Description Details (`dd`) element**: Used to represent the description of the term.
- **Block Quotation (`blockquote`) element**: Used to represent a section that is quoted from another source.
- **Inline Quotation (`q`) element**: Used to represent a short inline quotation.
- **Abbreviation (`abbr`) element**: Used to represent an abbreviation or acronym.
- **Contact Address (`address`) element**: Used to represent the contact information.
- **(Date) Time (`time`) element**: Used to represent a date and/or time.
- **Superscript (`sup`) element**: Used to represent superscript text.
- **Subscript (`sub`) element**: Used to represent subscript text.
- **Inline Code (`code`) element**: Used to represent a fragment of computer code.
- **Unarticulated Annotation (`u`) element**: Used to represent a span of inline text which should be rendered in a way that indicates that it has a non-textual annotation.
- **Ruby Annotation (`ruby`) element**: Used to represent the text of a ruby annotation.
- **Strikethrough (`s`) element**: Used to represent content that is no longer accurate or relevant.

HTML Form Elements and Attributes

Forms

- **`form` element**: Used to create an HTML form for user input.
- **`action` attribute**: Defines where to send form data.
- **`method` attribute**: Determines how form data is sent (`GET` or `POST`).
- **Common Input Types**:
  - `text`, `email`, `password`, `radio`, `checkbox`, `number`, `date`.
- **`action` attribute**: used to specify the URL where the form data should be sent.
- **`method` attribute**: used to specify the HTTP method to use when sending the form data. The most common methods are `GET` and `POST`.

```
<form method="value-goes-here" action="url-goes-here">
  <!-- inputs go inside here -->
</form>
```

- **`input` element**: used to create an input field for user input.
- **`type` attribute**: used to specify the type of input field. Ex. `text`, `email`, `number`, `radio`, `checkbox`, etc.
- **`placeholder` attribute**: used to show a hint to the user to show them what to enter in the input field.
- **`value` attribute**: used to specify the value of the input. If the input has a `button` type, the `value` attribute can be used to set the button text.
- **`name` attribute**: used to assign a name to an input field, which serves as the key when form data is submitted. For radio buttons, giving them the same `name` groups them together, so only one option in the group can be selected at a time.
- **`size` attribute**: used to define the number of characters that should be visible as the user types into the input.
- **`min` attribute**: can be used with input types such as `number` to specify the minimum value allowed in the input field.
- **`max` attribute**: can be used with input types such as `number` to specify the maximum value allowed in the input field.
- **`minlength` attribute**: used to specify the minimum number of characters required in an input field.
- **`maxlength` attribute**: used to specify the maximum number of characters allowed in an input field.
- **`required` attribute**: used to specify that an input field must be filled out before submitting the form.

- `disabled` **attribute**: used to specify that an input field should be disabled.
- `readonly` **attribute**: used to specify that an input field is read-only.

```
<!-- Text input -->
<input
  type="text"
  id="name"
  name="name"
  placeholder="e.g. Quincy Larson"
  size="20"
  minlength="5"
  maxlength="30"
  required
/>

<!-- Number input -->
<input
  type="number"
  id="quantity"
  name="quantity"
  min="2"
  max="10"
  disabled
/>

<!-- Button -->
<input type="button" value="Show Alert" />
```

- `label` **element**: used to create a label for an input field.
- `for` **attribute**: used to specify which input field the label is for.
- **Implicit form association**: inputs can be associated with labels by wrapping the input field inside the `label` element.

```
<form action="">
  <label>
    Full Name:
    <input type="text" />
  </label>
</form>
```

- **Explicit form association**: inputs can be associated with labels by using the `for` attribute on the `label` element.

```
<form action="">
  <label for="email">Email Address: </label>
  <input type="email" id="email" />
</form>
```

- `button` **element**: used to create a clickable button. A button can also have a `type` attribute, which is used to control the behavior of the button when it is activated. Ex. `submit`, `reset`, `button`.

```
<button type="button">Show Form</button>
<button type="submit">Submit Form</button>
<button type="reset">Reset Form</button>
```

- `fieldset` **element**: used to group related inputs together.

- `legend` **element**: used to add a caption to describe the group of inputs.

```
<!-- Radio group -->
<fieldset>
  <legend>Was this your first time at our hotel?</legend>

  <label for="yes-option">Yes</label>
  <input id="yes-option" type="radio" name="hotel-stay" value="yes" />

  <label for="no-option">No</label>
  <input id="no-option" type="radio" name="hotel-stay" value="no" />
</fieldset>

<!-- Checkbox group -->
<fieldset>
  <legend>
    Why did you choose to stay at our hotel? (Check all that apply)
  </legend>

  <label for="location">Location</label>
  <input type="checkbox" id="location" name="location" value="location" />

  <label for="price">Price</label>
  <input type="checkbox" id="price" name="price" value="price" />
</fieldset>
```

- **Focused state**: this is the state of an input field when it is selected by the user.

Working with HTML Table Elements and Attributes

- **Table element**: used to create an HTML table.
- **Table Head (`thead`) element**: used to group the header content in an HTML table.
- **Table Row (`tr`) element**: used to create a row in an HTML table.
- **Table Header (`th`) element**: used to create a header cell in an HTML table.
- **Table body (`tbody`) element**: used to group the body content in an HTML table.
- **Table Data Cell (`td`) element**: used to create a data cell in an HTML table.
- **Table Foot (`tfoot`) element**: used to group the footer content in an HTML table.
- `caption` **element**: used to add a title of an HTML table.
- `colspan` **attribute**: used to specify the number of columns a table cell should span.

```
<table>
  <caption>Exam Grades</caption>

  <thead>
    <tr>
      <th>Last Name</th>
      <th>First Name</th>
      <th>Grade</th>
    </tr>
  </thead>

  <tbody>
    <tr>
      <td>Davis</td>
      <td>Alex</td>
      <td>54</td>
    </tr>

    <tr>
      <td>Doe</td>
      <td>Samantha</td>
```

```
        <td>92</td>
      </tr>

      <tr>
        <td>Rodriguez</td>
        <td>Marcus</td>
        <td>88</td>
      </tr>
    </tbody>

    <tfoot>
      <tr>
        <td colspan="2">Average Grade</td>
        <td>78</td>
      </tr>
    </tfoot>
  </table>
```

HTML Tools

- **HTML validator**: A tool that checks the syntax of HTML code to ensure it is valid.
- **DOM inspector**: A tool that allows you to inspect and modify the HTML structure of a web page.
- **Devtools**: A set of web developer tools built directly into the browser that helps you debug, profile, and analyze web pages.

Introduction to Accessibility

- **Web Content Accessibility Guidelines**: The Web Content Accessibility Guidelines (WCAG) are a set of guidelines for making web content more accessible to people with disabilities. The four principles of WCAG are POUR which stands for Perceivable, Operable, Understandable, and Robust.

Assistive Technology for Accessibility

- **Screen readers**: Software programs that read the content of a computer screen out loud. They are used by people who are blind or visually impaired to access the web.
- **Large text or braille keyboards**: Used by people with visual impairments to access the web.
- **Screen magnifiers**: Software programs that enlarge the content of a computer screen. They are used by people with low vision to access the web.
- **Alternative pointing devices**: Used by people with mobility impairments to access the web. This includes devices such as joysticks, trackballs, and touchpads.
- **Voice recognition**: Used by people with mobility impairments to access the web. It allows users to control a computer using their voice.

Accessibility Auditing Tools

- **Common Accessibility Tools**: Google Lighthouse, Wave, IBM Equal Accessibility Checker, and axe DevTools are some of the common accessibility tools used to audit the accessibility of a website.

Accessibility Best Practices

- **Proper heading level structure**: You should use proper heading levels to create a logical structure for your content. This helps assistive technologies understand the content of your website.
- **Accessibility and Tables**: When using tables, you should use the `th` element to define header cells and the `td` element to define data cells. This helps assistive technologies understand the structure of the table.
- **Importance for inputs to have an associated label**: You should use the `label` element to associate labels with form inputs. This helps assistive technologies understand the purpose of the input.
- **Importance of good `alt` text**: You should use the `alt` attribute to provide a text alternative for images. This helps assistive technologies understand the content of the image.

- **Importance of good link text**: You should use descriptive link text to help users understand the purpose of the link. This helps assistive technologies understand the purpose of the link.
- **Best practices for making audio and video content accessible**: You should provide captions and transcripts for audio and video content to make it accessible to people with hearing impairments. You should also provide audio descriptions for video content to make it accessible to people with visual impairments.
- **`tabindex` attribute**: Used to make elements focusable and define the relative order in which they should be navigated using the keyboard. It is important to never use the `tabindex` attribute with a value greater than 0. Instead, you should either use a value of 0 or -1.
- **`accesskey` attribute**: Used to define a keyboard shortcut for an element. This can help users with mobility impairments navigate the website more easily.

WAI-ARIA, Roles, and Attributes

- **WAI-ARIA**: It stands for Web Accessibility Initiative - Accessible Rich Internet Applications. It is a set of attributes that can be added to HTML elements to improve accessibility. It provides additional information to assistive technologies about the purpose and structure of the content.
- **ARIA roles**: A set of predefined roles that can be added to HTML elements to define their purpose. This helps assistive technologies understand the content of the website. Examples include `role="tab"`, `role="menu"`, and `role="alert"`.
- **`aria-label` and `aria-labelledby` attributes**: These attributes are used to give an element a programmatic (or accessible) name, which helps assistive technology (such as screen readers) understand the purpose of the element. They are often used when the visual label for an element is an image or symbol rather than text. `aria-label` allows you to define the name directly in the attribute while `aria-labelledby` allows you to reference existing text on the page.
- **`aria-hidden` attribute**: Used to hide an element from assistive technologies such as screen readers. For example, this can be used to hide decorative images that do not provide any meaningful content.
- **`aria-describedby` attribute**: Used to provide additional information about an element by associating it with another element that contains the information. This helps assistive technologies understand the purpose of the element.

--assignment--

Review the HTML topics and concepts.

HTML Accessibility Review

Introduction to Accessibility

- **Web Content Accessibility Guidelines**: The Web Content Accessibility Guidelines (WCAG) are a set of guidelines for making web content more accessible to people with disabilities. The four principles of WCAG are POUR which stands for Perceivable, Operable, Understandable, and Robust.

Assistive Technology for Accessibility

- **Screen readers**: Software programs that read the content of a computer screen out loud. They are used by people who are blind or visually impaired to access the web.
- **Large text or braille keyboards**: Used by people with visual impairments to access the web.
- **Screen magnifiers**: Software programs that enlarge the content of a computer screen. They are used by people with low vision to access the web.
- **Alternative pointing devices**: Used by people with mobility impairments to access the web. This includes devices such as joysticks, trackballs, and touchpads.
- **Voice recognition**: Used by people with mobility impairments to access the web. It allows users to control a computer using their voice.

Accessibility Auditing Tools

- **Common Accessibility Tools**: Google Lighthouse, Wave, IBM Equal Accessibility Checker, and axe DevTools are some of the common accessibility tools used to audit the accessibility of a website.

Accessibility Best Practices

- **Proper heading level structure**: You should use proper heading levels to create a logical structure for your content. This helps people using assistive technologies understand the content of your website.
- **Accessibility and Tables**: When using tables, you should use the `th` element to define header cells and the `td` element to define data cells. This helps people using assistive technologies understand the structure of the table. With the `caption` element, you can write the caption (or title) of a table, so users, especially those who use assistive technologies, can quickly understand the table's purpose and content. You should place the `caption` element immediately after the opening tag of the `table` element. This way, screen readers and other assistive technologies can provide more context by announcing the caption before reading the content.
- **Importance for inputs to have an associated label**: You should use the `label` element to associate labels with form inputs. This helps people using assistive technologies understand the purpose of the input.
- **Importance of good `alt` text**: You should use the `alt` attribute to provide a text alternative for images. This helps people using assistive technologies understand the content of the image.
- **Importance of good link text**: You should use descriptive link text to help users understand the purpose of the link. This helps people using assistive technologies understand the purpose of the link.
- **Best practices for making audio and video content accessible**: You should provide captions and transcripts for audio and video content to make it accessible to people with hearing impairments. You should also provide audio descriptions for video content to make it accessible to people with visual impairments.
- **`tabindex` attribute**: Used to make elements focusable and define the relative order in which they should be navigated using the keyboard. It is important to never use the `tabindex` attribute with a value greater than 0. Instead, you should either use a value of 0 or -1.

```
<p tabindex="-1">Sorry, there was an error with your submission.</p>
```

- **`accesskey` attribute**: Used to define a keyboard shortcut for an element. This can help users with mobility impairments navigate the website more easily.

```
<button accesskey="s">Save</button>
<button accesskey="c">Cancel</button>
<a href="index.html" accesskey="h">Home</a>
```

WAI-ARIA, Roles, and Attributes

- **WAI-ARIA**: It stands for Web Accessibility Initiative - Accessible Rich Internet Applications. It is a set of attributes that can be added to HTML elements to improve accessibility. It provides additional information to assistive technologies about the purpose and structure of the content.
- **ARIA roles**: A set of predefined roles that can be added to HTML elements to define their purpose. This helps people using assistive technologies understand the content of the website. Examples include `role="tab"`, `role="menu"`, and `role="alert"`.

There are six main categories of ARIA roles:

- **Document structure roles**: These roles define the overall structure of the web page. With these roles, assistive technologies can understand the relationships between different sections and help

users navigate the content.

- **Widget roles**: These roles define the purpose and functionality of interactive elements, like scrollbars.
- **Landmark roles**: These roles classify and label the primary sections of a web page. Screen readers use them to provide convenient navigation to important sections of a page.
- **Live region roles**: These roles define elements with content that will change dynamically. This way, screen readers and other assistive technologies can announce changes to users with visual disabilities.
- **Window roles**: These roles define sub-windows, like pop up modal dialogues. These roles include `alertdialog` and `dialog`.
- **Abstract roles**: These roles help organize the document. They're only meant to be used internally by the browser, not by developers, so you should know that they exist but you shouldn't use them on your websites or web applications.
- **`aria-label` and `aria-labelledby` attributes**: These attributes are used to give an element a programmatic (or accessible) name, which helps people using assistive technology (such as screen readers) understand the purpose of the element. They are often used when the visual label for an element is an image or symbol rather than text. `aria-label` allows you to define the name directly in the attribute while `aria-labelledby` allows you to reference existing text on the page.

```
<button aria-label="Search">
    <i class="fas fa-search"></i>
</button>
```

```
<input type="text" aria-labelledby="search-btn">
<button type="button" id="search-btn">Search</button>
```

- **`aria-hidden` attribute**: Used to hide an element from assistive technologies such as screen readers. For example, this can be used to hide decorative images that do not provide any meaningful content.

```
<button>
    <i class="fa-solid fa-gear" aria-hidden="true"></i>
    <span class="label">Settings</span>
</button>
```

- **`aria-describedby` attribute**: Used to provide additional information about an element by associating it with another element that contains the information. This gives people using screen readers immediate access to the additional information when they navigate to the element. Common usage would include associating formatting instructions to a text input or an error message to an input after an invalid form submission.

```
<form>
    <label for="password">Password:</label>
    <input type="password" id="password" aria-describedby="password-help" />
    <p id="password-help">Your password must be at least 8 characters long.
</p>
</form>
```

--assignment--

Review the HTML Accessibility topics and concepts.

HTML Tables and Forms Review

HTML Form Elements and Attributes

- `form` **element**: used to create an HTML form for user input.
- `action` **attribute**: used to specify the URL where the form data should be sent.
- `method` **attribute**: used to specify the HTTP method to use when sending the form data. The most common methods are `GET` and `POST`.

```
<form method="value-goes-here" action="url-goes-here">
  <!-- inputs go inside here -->
</form>
```

- `input` **element**: used to create an input field for user input.
- `type` **attribute**: used to specify the type of input field. Ex. `text`, `email`, `number`, `radio`, `checkbox`, etc.
- `placeholder` **attribute**: used to show a hint to the user to show them what to enter in the input field.
- `value` **attribute**: used to specify the value of the input. If the input has a `button` type, the `value` attribute can be used to set the button text.
- `name` **attribute**: used to assign a name to an input field, which serves as the key when form data is submitted. For radio buttons, giving them the same `name` groups them together, so only one option in the group can be selected at a time.
- `size` **attribute**: used to define the number of characters that should be visible as the user types into the input.
- `min` **attribute**: can be used with input types such as `number` to specify the minimum value allowed in the input field.
- `max` **attribute**: can be used with input types such as `number` to specify the maximum value allowed in the input field.
- `minlength` **attribute**: used to specify the minimum number of characters required in an input field.
- `maxlength` **attribute**: used to specify the maximum number of characters allowed in an input field.
- `required` **attribute**: used to specify that an input field must be filled out before submitting the form.
- `disabled` **attribute**: used to specify that an input field should be disabled.
- `readonly` **attribute**: used to specify that an input field is read-only.

```
<!-- Text input -->
<input
  type="text"
  id="name"
  name="name"
  placeholder="e.g. Quincy Larson"
  size="20"
  minlength="5"
  maxlength="30"
  required
/>

<!-- Number input -->
<input
  type="number"
```

```
  id="quantity"
  name="quantity"
  min="2"
  max="10"
  disabled
/>

<!-- Button -->
<input type="button" value="Show Alert" />
```

- **`label` element**: used to create a label for an input field.
- **`for` attribute**: used to specify which input field the label is for.
- **Implicit form association**: inputs can be associated with labels by wrapping the input field inside the `label` element.

```
<form action="">
  <label>
    Full Name:
    <input type="text" />
  </label>
</form>
```

- **Explicit form association**: inputs can be associated with labels by using the `for` attribute on the `label` element.

```
<form action="">
  <label for="email">Email Address: </label>
  <input type="email" id="email" />
</form>
```

- **`button` element**: used to create a clickable button. A button can also have a `type` attribute, which is used to control the behavior of the button when it is activated. Ex. `submit`, `reset`, `button`.

```
<button type="button">Show Form</button>
<button type="submit">Submit Form</button>
<button type="reset">Reset Form</button>
```

- **`fieldset` element**: used to group related inputs together.
- **`legend` element**: used to add a caption to describe the group of inputs.

```
<!-- Radio group -->
<fieldset>
  <legend>Was this your first time at our hotel?</legend>

  <label for="yes-option">Yes</label>
  <input id="yes-option" type="radio" name="hotel-stay" value="yes" />

  <label for="no-option">No</label>
  <input id="no-option" type="radio" name="hotel-stay" value="no" />
</fieldset>

<!-- Checkbox group -->
<fieldset>
  <legend>
    Why did you choose to stay at our hotel? (Check all that apply)
  </legend>
```

```
  <label for="location">Location</label>
  <input type="checkbox" id="location" name="location" value="location" />

  <label for="price">Price</label>
  <input type="checkbox" id="price" name="price" value="price" />
</fieldset>
```

- **Focused state**: this is the state of an input field when it is selected by the user.

Working with HTML Table Elements and Attributes

- **Table element**: used to create an HTML table.
- **Table Head (`thead`) element**: used to group the header content in an HTML table.
- **Table Row (`tr`) element**: used to create a row in an HTML table.
- **Table Header (`th`) element**: used to create a header cell in an HTML table.
- **Table body (`tbody`) element**: used to group the body content in an HTML table.
- **Table Data Cell (`td`) element**: used to create a data cell in an HTML table.
- **Table Foot (`tfoot`) element**: used to group the footer content in an HTML table.
- `caption` **element**: used to add a title of an HTML table.
- `colspan` **attribute**: used to specify the number of columns a table cell should span.

```
<table>
  <caption>Exam Grades</caption>

  <thead>
    <tr>
      <th>Last Name</th>
      <th>First Name</th>
      <th>Grade</th>
    </tr>
  </thead>

  <tbody>
    <tr>
      <td>Davis</td>
      <td>Alex</td>
      <td>54</td>
    </tr>

    <tr>
      <td>Doe</td>
      <td>Samantha</td>
      <td>92</td>
    </tr>

    <tr>
      <td>Rodriguez</td>
      <td>Marcus</td>
      <td>88</td>
    </tr>
  </tbody>

  <tfoot>
    <tr>
      <td colspan="2">Average Grade</td>
      <td>78</td>
    </tr>
  </tfoot>
</table>
```

Working with HTML Tools

- **HTML validator**: a tool that checks the syntax of HTML code to ensure it is valid.
- **DOM inspector**: a tool that allows you to inspect and modify the HTML structure of a web page.
- **Devtools**: a set of web developer tools built directly into the browser that helps you debug, profile, and analyze web pages.

--assignment--

Review the HTML Tables and Forms topics and concepts.

JavaScript Review

Review the concepts below to prepare for the upcoming prep exam.

Working with HTML, CSS, and JavaScript

While HTML and CSS provide website structure, JavaScript brings interactivity to websites by enabling complex functionality, such as handling user input, animating elements, and even building full web applications.

Data Types in JavaScript

Data types help the program understand the kind of data it's working with, whether it's a number, text, or something else.

- **Number**: A number represents both integers and floating-point values. Examples of integers include 7, 19, and 90.
- **Floating point**: A floating point number is a number with a decimal point. Examples include 3.14, 0.5, and 0.0001.
- **String**: A string is a sequence of characters, or text, enclosed in quotes. `"I like coding"` and `'JavaScript is fun'` are examples of strings.
- **Boolean**: A boolean represents one of two possible values: `true` or `false`. You can use a boolean to represent a condition, such as `isLoggedIn = true`.
- **Undefined and Null**: An undefined value is a variable that has been declared but not assigned a value. A null value is an empty value or a variable that has intentionally been assigned a value of `null`.
- **Object**: An object is a collection of key-value pairs. The key is the property name, and the value is the property value.

Here, the `pet` object has three properties or keys: `name`, `age`, and `type`. The values are `Fluffy`, `3`, and `dog`, respectively.

```
let pet = {
  name: 'Fluffy',
  age: 3,
  type: 'dog'
};
```

- **Symbol**: The Symbol data type is a unique and immutable value that may be used as an identifier for object properties.

In the example below, two symbols are created with the same description, but they are not equal.

```
```

```
const crypticKey1= Symbol('saltNpepper');
const crypticKey2= Symbol('saltNpepper');
console.log(crypticKey1 === crypticKey2); // false
```

- **BigInt**: When the number is too large for the `Number` data type, you can use the BigInt data type to represent integers of arbitrary length.

By adding an `n` to the end of the number, you can create a BigInt.

```
const veryBigNumber = 12345678901234567890123456789012345678901234567890n;
```

Variables in JavaScript

- Variables can be declared using the `let` keyword.

```
let cityName;
```

- To assign a value to a variable, you can use the assignment operator `=`.

```
cityName = 'New York';
```

- Variables declared using `let` can be reassigned a new value.

```
cityName = 'Los Angeles';
console.log(cityName); // Los Angeles
```

- Apart from `let`, you can also use `const` to declare a variable. However, a `const` variable cannot be reassigned a new value.

```
const cityName = 'New York';
cityName = 'Los Angeles'; // TypeError: Assignment to constant variable.
```

- Variables declared using `const` find uses in declaring constants, that are not allowed to change throughout the code, such as `PI` or `MAX_SIZE`.

Variable Naming Conventions

- Variable names should be descriptive and meaningful.
- Variable names should be camelCase like `cityName`, `isLoggedIn`, and `veryBigNumber`.
- Variable names should not start with a number. They must begin with a letter, `_`, or `$`.
- Variable names should not contain spaces or special characters, except for `_` and `$`.
- Variable names should not be reserved keywords.
- Variable names are case-sensitive. `age` and `Age` are different variables.

Strings and String immutability in JavaScript

- Strings are sequences of characters enclosed in quotes. They can be created using single quotes and double quotes.

```

```

```
let correctWay = 'This is a string';
let alsoCorrect = "This is also a string";
```

- Strings are immutable in JavaScript. This means that once a string is created, you cannot change the characters in the string. However, you can still reassign strings to a new value.

```
let firstName = 'John';
firstName = 'Jane'; // Reassigning the string to a new value
```

String Concatenation in JavaScript

- Concatenation is the process of joining multiple strings or combining strings with variables that hold text. The `+` operator is one of the simplest and most frequently used methods to concatenate strings.

```
let studentName = 'Asad';
let studentAge = 25;
let studentInfo = studentName + ' is ' + studentAge + ' years old.';
console.log(studentInfo); // Asad is 25 years old.
```

- If you need to add or append to an existing string, then you can use the `+=` operator. This is helpful when you want to build upon a string by adding more text to it over time.

```
let message = 'Welcome to programming, ';
message += 'Asad!';
console.log(message); // Welcome to programming, Asad!
```

- Another way you can concatenate strings is to use the `concat()` method. This method joins two or more strings together.

```
let firstName = 'John';
let lastName = 'Doe';
let fullName = firstName.concat(' ', lastName);
console.log(fullName); // John Doe
```

Logging Messages with `console.log()`

- The `console.log()` method is used to log messages to the console. It's a helpful tool for debugging and testing your code.

```
console.log('Hello, World!');
// Output: Hello, World!
```

Semicolons in JavaScript

- Semicolons are primarily used to mark the end of a statement. This helps the JavaScript engine understand the separation of individual instructions, which is crucial for correct execution.

```
let message = 'Hello, World!'; // first statement ends here
let number = 42; // second statement starts here
```

- Semicolons help prevent ambiguities in code execution and ensure that statements are correctly terminated.

Comments in JavaScript

- Any line of code that is commented out is ignored by the JavaScript engine. Comments are used to explain code, make notes, or temporarily disable code.
- Single-line comments are created using `//`.

```
// This is a single-line comment and will be ignored by the JavaScript engine
```

- Multi-line comments are created using `/*` to start the comment and `*/` to end the comment.

```
/*
This is a multi-line comment.
It can span multiple lines.
*/
```

JavaScript as a Dynamically Typed Language

- JavaScript is a dynamically typed language, which means that you don't have to specify the data type of a variable when you declare it. The JavaScript engine automatically determines the data type based on the value assigned to the variable.

```
let error = 404; // JavaScript treats error as a number
error = "Not Found"; // JavaScript now treats error as a string
```

- Other languages, like Java, that are not dynamically typed would result in an error:

```
int error = 404; // value must always be an integer
error = "Not Found"; // This would cause an error in Java
```

Using the `typeof` Operator

- The `typeof` operator is used to check the data type of a variable. It returns a string indicating the type of the variable.

```
let age = 25;
console.log(typeof age); // number

let isLoggedIn = true;
console.log(typeof isLoggedIn); // boolean
```

- However, there's a well-known quirk in JavaScript when it comes to null. The `typeof` operator returns `object` for null values.

```
let user = null;
console.log(typeof user); // object
```

String Basics

- **Definition**: A string is a sequence of characters wrapped in either single quotes, double quotes or backticks. Strings are primitive data types and they are immutable. Immutability means that once a string is created, it cannot be changed.
- **Accessing Characters from a String**: To access a character from a string you can use bracket notation and pass in the index number. An index is the position of a character within a string, and it is zero-based.

```
const developer = "Jessica";
developer[0] // J
```

- `\n` **(Newline Character)**: You can create a newline in a string by using the `\n` newline character.

```
const poem = "Roses are red,\nViolets are blue,\nJavaScript is fun,\nAnd so
are you.";
console.log(poem);
```

- **Escaping Strings**: You can escape characters in a string by placing backlashes (`\`) in front of the quotes.

```
const statement = "She said, \"Hello!\"";
console.log(statement); // She said, "Hello!"
```

Template Literals (Template Strings) and String Interpolation

- **Definition**: Template literals are defined with backticks (`). They allow for easier string manipulation, including embedding variables directly inside a string, a feature known as string interpolation.

```
const name = "Jessica";
const greeting = `Hello, ${name}!`; // "Hello, Jessica!"
```

ASCII, the `charCodeAt()` Method and the `fromCharCode()` Method

- **ASCII**: ASCII, short for American Standard Code for Information Interchange, is a character encoding standard used in computers to represent text. It assigns a numeric value to each character, which is universally recognized by machines.
- **The `charCodeAt()` Method**: This method is called on a string and returns the ASCII code of the character at a specified index.

```
const letter = "A";
console.log(letter.charCodeAt(0));  // 65
```

- **The `fromCharCode()` Method**: This method converts an ASCII code into its corresponding character.

```
const char = String.fromCharCode(65);
console.log(char);  // A
```

Other Common String Methods

- **The `indexOf` Method**: This method is used to search for a substring within a string. If the substring is found, `indexOf` returns the index (or position) of the first occurrence of that substring. If the substring is not found, `indexOf` returns -1, which indicates that the search was unsuccessful.

```
const text = "The quick brown fox jumps over the lazy dog.";
console.log(text.indexOf("fox")); // 16
console.log(text.indexOf("cat")); // -1
```

- **The `includes()` Method**: This method is used to check if a string contains a specific substring. If the substring is found within the string, the method returns true. Otherwise, it returns false.

```
const text = "The quick brown fox jumps over the lazy dog.";
console.log(text.includes("fox")); // true
console.log(text.includes("cat")); // false
```

- The **`slice()` Method**: This method returns a new array containing a shallow copy of a portion of the original array, specified by start and end indices. The new array contains references to the same elements as the original array (not duplicates). This means that if the elements are primitives (like numbers or strings), the values are copied; but if the elements are objects or arrays, the references are copied, not the objects themselves.

```
const text = "freeCodeCamp";
console.log(text.slice(0, 4));  // "free"
console.log(text.slice(4, 8));  // "Code"
console.log(text.slice(8, 12)); // "Camp"
```

- **The `toUpperCase()` Method**: This method converts all the characters to uppercase letters and returns a new string with all uppercase characters.

```
const text = "Hello, world!";
console.log(text.toUpperCase()); // "HELLO, WORLD!"
```

- **The `toLowerCase()` Method**: This method converts all characters in a string to lowercase.

```
const text = "HELLO, WORLD!"
console.log(text.toLowerCase()); // "hello, world!"
```

- **The `replace()` Method**: This method allows you to find a specified value (like a word or character) in a string and replace it with another value. The method returns a new string with the replacement and leaves the original unchanged because JavaScript strings are immutable.

```
const text = "I like cats";
console.log(text.replace("cats", "dogs")); // "I like dogs"
```

- **The `repeat()` Method**: This method is used to repeat a string a specified number of times.

```

```

```
const text = "Hello";
console.log(text.repeat(3)); // "HelloHelloHello"
```

- **The `trim()` Method**: This method is used to remove whitespaces from both the beginning and the end of a string.

```
const text = "  Hello, world!  ";
console.log(text.trim()); // "Hello, world!"
```

- **The `trimStart()` Method**: This method removes whitespaces from the beginning (or "start") of the string.

```
const text = "  Hello, world!  ";
console.log(text.trimStart()); // "Hello, world!  "
```

- **The `trimEnd()` Method**: This method removes whitespaces from the end of the string.

```
const text = " Hello, world! ";
console.log(text.trimEnd()); // "  Hello, world!"
```

- **The `prompt()` Method**: This method of the `window` is used to get information from a user through the form of a dialog box. This method takes two arguments. The first argument is the message which will appear inside the dialog box, typically prompting the user to enter information. The second one is a default value which is optional and will fill the input field initially.

```
const answer = window.prompt("What's your favorite animal?"); // This will
change depending on what the user answers
```

Working with the Number Data Type

- **Definition**: JavaScript's `Number` type includes integers, floating-point numbers, `Infinity` and `NaN`. Floating-point numbers are numbers with a decimal point. Positive `Infinity` is a number greater than any other number while `-Infinity` is a number smaller than any other number. `NaN` (`Not a Number`) represents an invalid numeric value like the string `"Jessica"`.

Common Arithmetic Operations

- **Addition Operator**: This operator (`+`) is used to calculate the sum of two or more numbers.
- **Subtraction Operator**: This operator (`-`) is used to calculate the difference between two numbers.
- **Multiplication Operator**: This operator (`*`) is used to calculate the product of two or more numbers.
- **Division Operator**: This operator (`/`) is used to calculate the quotient between two numbers
- **Division By Zero**: If you try to divide by zero, JavaScript will return `Infinity`.
- **Remainder Operator**: This operator(`%`) returns the remainder of a division.
- **Exponentiation Operator**: This operator (`**`) raises one number to the power of another.

Calculations with Numbers and Strings

- **Explanation**: When you use the `+` operator with a number and a string, JavaScript will coerce the number into a string and concatenate the two values. When you use the `-`, `*` or `/` operators with a string and number, JavaScript will coerce the string into a number and the result will be a number.

For `null` and `undefined`, JavaScript treats `null` as 0 and undefined as `NaN` in mathematical operations.

```
const result = 5 + '10';

console.log(result); // 510
console.log(typeof result); // string

const subtractionResult = '10' - 5;
console.log(subtractionResult); // 5
console.log(typeof subtractionResult); // number

const multiplicationResult = '10' * 2;
console.log(multiplicationResult); // 20
console.log(typeof multiplicationResult); // number

const divisionResult = '20' / 2;
console.log(divisionResult); // 10
console.log(typeof divisionResult); // number

const result1 = null + 5;
console.log(result1); // 5
console.log(typeof result1); // number

const result2 = undefined + 5;
console.log(result2); // NaN
console.log(typeof result2); // number
```

Operator Precedence

- **Definition**: Operator precedence determines the order in which operations are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. Values inside the parenthesis will be evaluated first and multiplication/division will have higher precedence than addition/subtraction. If the operators have the same precedence, then JavaScript will use associativity. Associativity is what tells JavaScript whether to evaluate operators from left to right or right to left. For example, the exponent operator is also right to left associative:

```
const result = (2 + 3) * 4;

console.log(result); // 20

const result2 = 10 - 2 + 3;

console.log(result2); // 11

const result3 = 2 ** 3 ** 2;

console.log(result3); // 512
```

Increment and Decrement Operators

- **Increment Operator**: This operator is used to increase the value by one. The prefix notation `++num` increases the value of the variable first, then returns a new value. The postfix notation `num++` returns the current value of the variable first, then increases it.

```
let x = 5;

console.log(++x); // 6
```

```
console.log(x); // 6


let y = 5;

console.log(y++); // 5
console.log(y); // 6
```

- **Decrement Operator**: This operator is used to decrease the value by one. The prefix and postfix notation works the same way as earlier with the increment operator.

```
let num = 5;

console.log(--num); // 4
console.log(num--); // 4
console.log(num); // 3
```

Compound Assignment Operators

- **Addition Assignment (`+=`) Operator**: This operator performs addition on the values and assigns the result to the variable.
- **Subtraction Assignment (`-=`) Operator**: This operator performs subtraction on the values and assigns the result to the variable.
- **Multiplication Assignment (`*=`) Operator**: This operator performs multiplication on the values and assigns the result to the variable.
- **Division Assignment (`/=`) Operator**: This operator performs division on the values and assigns the result to the variable.
- **Remainder Assignment (`%=`) Operator**: This operator divides a variable by the specified number and assigns the remainder to the variable.
- **Exponentiation Assignment (`**=`) Operator**: This operator raises a variable to the power of the specified number and reassigns the result to the variable.

Booleans and Equality

- **Boolean Definition**: A boolean is a data type that can only have two values: `true` or `false`.
- **Equality (`==`) Operator**: This operator uses type coercion before checking if the values are equal.

```
console.log(5 == '5'); // true
```

- **Strict Equality (`===`) Operator**: This operator does not perform type coercion and checks if both the types and values are equal.

```
console.log(5 === '5'); // false
```

- **Inequality (`!=`) Operator**: This operator uses type coercion before checking if the values are not equal.
- **Strict Inequality (`!==`) Operator**: This operator does not perform type coercion and checks if both the types and values are not equal.

Comparison Operators

- **Greater Than (`>`) Operator**: This operator checks if the value on the left is greater than the one on the right.

- **Greater Than (`>=`) or Equal Operator**: This operator checks if the value on the left is greater than or equal to the one on the right.
- **Less Than (`<`) Operator**: This operator checks if the value on the left is less than the one on the right.
- **Less Than (`<=`) or Equal Operator**: This operator checks if the value on the left is less than or equal to the one on the right.

Unary Operators

- **Unary Plus Operator**: This operator converts its operand into a number. If the operand is already a number, it remains unchanged.

```
const str = '42';
const num = +str;

console.log(num); // 42
console.log(typeof num); // number
```

- **Unary Negation (`-`) Operator**: This operator negates the operand.

```
const num = 4;
console.log(-num); // -4
```

- **Logical NOT (`!`) Operator**: This operator flips the boolean value of its operand. So, if the operand is `true`, it becomes `false`, and if it's `false`, it becomes `true`.

Bitwise Operators

- **Bitwise AND (`&`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of both operands are 1.
- **Bitwise AND Assignment (`&=`) Operator**: This operator performs a `bitwise AND` operation with the specified number and reassigns the result to the variable.
- **Bitwise OR (`|`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of either or both operands are 1.
- **Bitwise OR Assignment (`|=`) Operator**: This operator performs a `bitwise OR` operation with the specified number and reassigns the result to the variable.
- **Bitwise XOR (`^`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of either, but not both, operands are 1.
- **Bitwise NOT (`~`) Operator**: This operator inverts the binary representation of a number.
- **Left Shift (`<<`) Operator**: This operator shifts all bits to the left by a specified number of positions.
- **Right Shift (`>>`) Operator**: This operator shifts all bits to the right.

Conditional Statements, Truthy Values, Falsy Values and the Ternary Operator

- **`if/else if/else`**: An `if` statement takes a condition and runs a block of code if that condition is `truthy`. If the condition is `false`, then it moves to the `else if` block. If none of those conditions are `true`, then it will execute the `else` clause. `Truthy` values are any values that result in `true` when evaluated in a Boolean context like an `if` statement. `Falsy` values are values that evaluate to `false` in a Boolean context.

```
const score = 87;

if (score >= 90) {
 console.log('You got an A');
} else if (score >= 80) {
```

```
 console.log('You got a B'); // You got an B
} else if (score >= 70) {
 console.log('You got a C');
} else {
 console.log('You failed! You need to study more!');
}
```

- **Ternary Operator**: This operator is often used as a shorter way to write `if else` statements.

```
const temperature = 30;
const weather = temperature > 25 ? 'sunny' : 'cool';

console.log(`It's a ${weather} day!`); // It's a sunny day!
```

Binary Logical Operators

- **Logical AND (`&&`) Operator**: This operator checks if both operands are truthy. If the first value is truthy, then it will return the second value. If the first value is falsy, then it will return the first value.

```
const result = true && 'hello';

console.log(result); // hello
```

- **Logical OR (`||`) Operator**: This operator checks if at least one of the operands is truthy. If the first value is truthy, then it is returned. If the first value is falsy, then the second value is returned.
- **Nullish Coalescing (`??`) Operator**: This operator will return a value only if the first one is `null` or `undefined`.

```
const userSettings = {
 theme: null,
 volume: 0,
 notifications: false,
};

let theme = userSettings.theme ?? 'light';
console.log(theme); // light
```

The `Math` Object

- **The `Math.random()` Method**: This method generates a random floating-point number between 0 (inclusive) and 1 (exclusive). This means the possible output can be 0, but it will never actually reach 1.
- **The `Math.max()` Method**: This method takes a set of numbers and returns the maximum value.
- **The `Math.min()` Method**: This method takes a set of numbers and returns the minimum value.
- **The `Math.ceil()` Method**: This method rounds a value up to the nearest whole integer.
- **The `Math.floor()` Method**: This method rounds a value down to the nearest whole integer.
- **The `Math.round()` Method**: This method rounds a value to the nearest whole integer.

```
console.log(Math.round(2.3)); // 2
console.log(Math.round(4.5)); // 5
console.log(Math.round(4.8)); // 5
```

- **The `Math.trunc()` Method**: This method removes the decimal part of a number, returning only the integer portion, without rounding.
- **The `Math.sqrt()` Method**: This method will return the square root of a number.
- **The `Math.cbrt()` Method**: This method will return the cube root of a number.
- **The `Math.abs()` Method**: This method will return the absolute value of a number.
- **The `Math.pow()` Method**: This method takes two numbers and raises the first to the power of the second.

Common Number Methods

- **`isNaN()`**: `NaN` stands for "Not-a-Number". It's a special value that represents an unrepresentable or undefined numerical result. The `isNaN()` function property is used to determine whether a value is `NaN` or not. `Number.isNaN()` provides a more reliable way to check for `NaN` values, especially in cases where type coercion might lead to unexpected results with the global `isNaN()` function.

```
console.log(isNaN(NaN));       // true
console.log(isNaN(undefined)); // true
console.log(isNaN({}));        // true

console.log(isNaN(true));      // false
console.log(isNaN(null));      // false
console.log(isNaN(37));        // false


console.log(Number.isNaN(NaN));        // true
console.log(Number.isNaN(Number.NaN)); // true
console.log(Number.isNaN(0 / 0));      // true

console.log(Number.isNaN("NaN"));      // false
console.log(Number.isNaN(undefined));  // false
```

- **The `parseFloat()` Method**: This method parses a string argument and returns a floating-point number. It's designed to extract a number from the beginning of a string, even if the string contains non-numeric characters later on.
- **The `parseInt()` Method**: This method parses a string argument and returns an integer. `parseInt()` stops parsing at the first non-digit it encounters. For floating-point numbers, it returns only the integer part. If it can't find a valid integer at the start of the string, it returns `NaN`.
- **The `toFixed()` Method**: This method is called on a number and takes one optional argument, which is the number of digits to appear after the decimal point. It returns a string representation of the number with the specified number of decimal places.

Comparisons and the `null` and `undefined` Data Types

- **Comparisons and `undefined`**: A variable is `undefined` when it has been declared but hasn't been assigned a value. It's the default value of uninitialized variables and function parameters that weren't provided an argument. `undefined` converts to `NaN` in numeric contexts, which makes all numeric comparisons with `undefined` return `false`.

```
console.log(undefined > 0);  // false
console.log(undefined < 0);  // false
console.log(undefined == 0); // false
```

- **Comparisons and `null`**: The `null` type represents the intentional absence of a value. When using the equality operator, `null` and `undefined` are considered equal. However, when using the strict equality operator (`===`), which checks both value and type without performing type coercion, `null` and `undefined` are not equal:

```
console.log(null == undefined); // true
console.log(null === undefined); // false
```

`switch` Statements

- **Definition**: A `switch` statement evaluates an expression and matches its value against a series of `case` clauses. When a match is found, the code block associated with that case is executed.

```
const dayOfWeek = 3;

switch (dayOfWeek) {
  case 1:
    console.log("It's Monday! Time to start the week strong.");
    break;
  case 2:
    console.log("It's Tuesday! Keep the momentum going.");
    break;
  case 3:
    console.log("It's Wednesday! We're halfway there.");
    break;
  case 4:
    console.log("It's Thursday! Almost the weekend.");
    break;
  case 5:
    console.log("It's Friday! The weekend is near.");
    break;
  case 6:
    console.log("It's Saturday! Enjoy your weekend.");
    break;
  case 7:
    console.log("It's Sunday! Rest and recharge.");
    break;
  default:
    console.log("Invalid day! Please enter a number between 1 and 7.");
}
```

JavaScript Functions

- Functions are reusable blocks of code that perform a specific task.
- Functions can be defined using the `function` keyword followed by a name, a list of parameters, and a block of code that performs the task.
- Arguments are values passed to a function when it is called.
- When a function finishes its execution, it will always return a value.
- By default, the return value of a function is `undefined`.
- The `return` keyword is used to specify the value to be returned from the function and ends the function execution.

Arrow Functions

- Arrow functions are a more concise way to write functions in JavaScript.
- Arrow functions are defined using the `=>` syntax between the parameters and the function body.
- When defining an arrow function, you do not need the `function` keyword.
- If you are using a single parameter, you can omit the parentheses around the parameter list.
- If the function body consists of a single expression, you can omit the curly braces and the `return` keyword.

Scope in Programming

- **Global scope**: This is the outermost scope in JavaScript. Variables declared in the global scope are accessible from anywhere in the code and are called global variables.
- **Local scope**: This refers to variables declared within a function. These variables are only accessible within the function where they are declared and are called local variables.
- **Block scope**: A block is a set of statements enclosed in curly braces `{}` such as in `if` statements, or loops.
- Block scoping with `let` and `const` provides even finer control over variable accessibility, helping to prevent errors and make your code more predictable.

JavaScript Array Basics

- **Definition**: A JavaScript array is an ordered collection of values, each identified by a numeric index. The values in a JavaScript array can be of different data types, including numbers, strings, booleans, objects, and even other arrays. Arrays are contiguous in memory, which means that all elements are stored in a single, continuous block of memory locations, allowing for efficient indexing and fast access to elements by their index.

```
const developers = ["Jessica", "Naomi", "Tom"];
```

- **Accessing Elements From Arrays**: To access elements from an array, you will need to reference the array followed by its index number inside square brackets. JavaScript arrays are zero based indexed which means the first element is at index 0, the second element is at index 1, etc. If you try to access an index that doesn't exist for the array, then JavaScript will return `undefined`.

```
const developers = ["Jessica", "Naomi", "Tom"];
developers[0] // "Jessica"
developers[1] // "Naomi"

developers[10] // undefined
```

- `length` **Property**: This property is used to return the number of items in an array.

```
const developers = ["Jessica", "Naomi", "Tom"];
developers.length // 3
```

- **Updating Elements in an Array**: To update an element in an array, you use the assignment operator (`=`) to assign a new value to the element at a specific index.

```
const fruits = ['apple', 'banana', 'cherry'];
fruits[1] = 'blueberry';

console.log(fruits); // ['apple', 'blueberry', 'cherry']
```

Two Dimensional Arrays

- **Definition**: A two-dimensional array is essentially an array of arrays. It's used to represent data that has a natural grid-like structure, such as a chessboard, a spreadsheet, or pixels in an image. To access an element in a two-dimensional array, you need two indices: one for the row and one for the column.

```
const chessboard = [
    ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
```

```
    ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
    ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']
];

console.log(chessboard[0][3]); // "Q"
```

Array Destructuring

- **Definition**: Array destructuring is a feature in JavaScript that allows you to extract values from arrays and assign them to variables in a more concise and readable way. It provides a convenient syntax for unpacking array elements into distinct variables.

```
const fruits = ["apple", "banana", "orange"];

const [first, second, third] = fruits;

console.log(first); // "apple"
console.log(second); // "banana"
console.log(third); // "orange"
```

- **Rest Syntax**: This allows you to capture the remaining elements of an array that haven't been destructured into a new array.

```
const fruits = ["apple", "banana", "orange", "mango", "kiwi"];
const [first, second, ...rest] = fruits;

console.log(first); // "apple"
console.log(second); // "banana"
console.log(rest); // ["orange", "mango", "kiwi"]
```

Common Array Methods

- `push()` **Method**: This method is used to add elements to the end of the array and will return the new length.

```
const desserts = ["cake", "cookies", "pie"];
desserts.push("ice cream");

console.log(desserts); // ["cake", "cookies", "pie", "ice cream"];
```

- `pop()` **Method**: This method is used to remove the last element from an array and will return that removed element. If the array is empty, then the return value will be `undefined`.

```
const desserts = ["cake", "cookies", "pie"];
desserts.pop();

console.log(desserts); // ["cake", "cookies"];
```

- `shift()` **Method**: This method is used to remove the first element from an array and return that removed element. If the array is empty, then the return value will be `undefined`.

```
const desserts = ["cake", "cookies", "pie"];
desserts.shift();

console.log(desserts); // ["cookies", "pie"];
```

- `unshift()` **Method**: This method is used to add elements to the beginning of the array and will return the new length.

```
const desserts = ["cake", "cookies", "pie"];
desserts.unshift("ice cream");

console.log(desserts); // ["ice cream", "cake", "cookies", "pie"];
```

- `indexOf()` **Method**: This method is useful for finding the first index of a specific element within an array. If the element cannot be found, then it will return `-1`.

```
const fruits = ["apple", "banana", "orange", "banana"];
const index = fruits.indexOf("banana");

console.log(index); // 1
console.log(fruits.indexOf("not found")); // -1
```

- `splice()` **Method**: This method is used to add or remove elements from any position in an array. The return value for the `splice()` method will be an array of the items removed from the array. If nothing was removed, then an empty array will be returned. This method will mutate the original array, modifying it in place rather than creating a new array. The first argument specifies the index at which to begin modifying the array. The second argument are the number of elements you wish to remove. The following arguments are the elements you wish to add.

```
const colors = ["red", "green", "blue"];
colors.splice(1, 0, "yellow", "purple");

console.log(colors); // ["red", "yellow", "purple", "green", "blue"]
```

- `includes()` **Method**: This method is used to check if an array contains a specific value. This method returns `true` if the array contains the specified element, and `false` otherwise.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];

console.log(programmingLanguages.includes("Python")); // true
console.log(programmingLanguages.includes("Perl")); // false
```

- `concat()` **Method**: This method creates a new array by merging two or more arrays.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];
const newList = programmingLanguages.concat("Perl");

console.log(newList); // ["JavaScript", "Python", "C++", "Perl"]
```

- `slice()` **Method**: This method returns a shallow copy of a portion of the array, starting from a specified index or the entire array. A shallow copy will copy the reference to the array instead of duplicating it.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];
const newList = programmingLanguages.slice(1);

console.log(newList); // ["Python", "C++"]
```

- **Spread Syntax**: The spread syntax is used to create shallow copies of an array.

```
const originalArray = [1, 2, 3];
const shallowCopiedArray = [...originalArray];

shallowCopiedArray.push(4);

console.log(originalArray); // [1, 2, 3]
console.log(shallowCopiedArray); // [1, 2, 3, 4]
```

- `split()` **Method**: This method divides a string into an array of substrings and specifies where each split should happen based on a given separator. If no separator is provided, the method returns an array containing the original string as a single element.

```
const str = "hello";
const charArray = str.split("");

console.log(charArray); // ["h", "e", "l", "l", "o"]
```

- `reverse()` **Method**: This method reverses an array in place.

```
const desserts = ["cake", "cookies", "pie"];
console.log(desserts.reverse()); // ["pie", "cookies", "cake"]
```

- `join()` **Method**: This method concatenates all the elements of an array into a single string, with each element separated by a specified separator. If no separator is provided, or an empty string (`""`) is used, the elements will be joined without any separator.

```
const reversedArray = ["o", "l", "l", "e", "h"];
const reversedString = reversedArray.join("");

console.log(reversedString); // "olleh"
```

Object Basics

- **Definition**: An object is a data structure that is made up of properties. A property consists of a key and a value. To access data from an object you can use either dot notation or bracket notation.

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York"
};
```

```
console.log(person.name);  // Alice
console.log(person["name"]); // Alice
```

To set a property of an existing object you can use either dot notation or bracket notation together with the assignment operator.

```
const person = {
  name: "Alice",
  age: 30
};

person.job = "Engineer"
person["hobby"] = "Knitting"
console.log(person);  // {name: 'Alice', age: 30, job: 'Engineer', hobby:
'Knitting'}
```

Removing Properties From an Object

- `delete` **Operator**: This operator is used to remove a property from an object.

```
const person = {
  name: "Alice",
  age: 30,
  job: "Engineer"
};

delete person.job;

console.log(person.job); // undefined
```

Checking if an Object has a Property

- `hasOwnProperty()` **Method**: This method returns a boolean indicating whether the object has the specified property as its own property.

```
const person = {
  name: "Alice",
  age: 30
};

console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("job")); // false
```

- `in` **Operator**: This operator will return `true` if the property exists in the object.

```
const person = {
  name: "Bob",
  age: 25
};

console.log("name" in person);  // true
```

Accessing Properties From Nested Objects

- **Accessing Data**: Accessing properties from nested objects involves using the dot notation or bracket notation, much like accessing properties from simple objects. However, you'll need to chain these accessors to drill down into the nested structure.

```
const person = {
  name: "Alice",
  age: 30,
  contact: {
    email: "alice@example.com",
    phone: {
      home: "123-456-7890",
      work: "098-765-4321"
    }
  }
};

console.log(person.contact.phone.work); // "098-765-4321"
```

Primitive and Non Primitive Data Types

- **Primitive Data Types**: These data types include numbers, strings, booleans, `null`, `undefined`, and symbols. These types are called "primitive" because they represent single values and are not objects. Primitive values are immutable, which means once they are created, their value cannot be changed.
- **Non Primitive Data Types**: In JavaScript, these are objects, which include regular objects, arrays, and functions. Unlike primitives, non-primitive types can hold multiple values as properties or elements.

Object Methods

- **Definition**: Object methods are functions that are associated with an object. They are defined as properties of an object and can access and manipulate the object's data. The `this` keyword inside the method refers to the object itself, enabling access to its properties.

```
const person = {
  name: "Bob",
  age: 30,
  sayHello: function() {
    return "Hello, my name is " + this.name;
  }
};

console.log(person.sayHello()); // "Hello, my name is Bob"
```

Object Constructor

- **Definition**: In JavaScript, a constructor is a special type of function used to create and initialize objects. It is invoked with the `new` keyword and can initialize properties and methods on the newly created object. The `Object()` constructor creates a new empty object.

```
new Object()
```

Working with the Optional Chaining Operator (`?.`)

- **Definition**: This operator lets you safely access object properties or call methods without worrying whether they exist.

```
const user = {
  name: "John",
  profile: {
    email: "john@example.com",
    address: {
      street: "123 Main St",
      city: "Somewhere"
    }
  }
};

console.log(user.profile?.address?.street); // "123 Main St"
console.log(user.profile?.phone?.number);   // undefined
```

Object Destructuring

- **Definition**: Object destructuring allows you to extract values from objects and assign them to variables in a more concise and readable way.

```
const person = { name: "Alice", age: 30, city: "New York" };

const { name, age } = person;

console.log(name); // Alice
console.log(age);  // 30
```

Working with JSON

- **Definition**: JSON stands for JavaScript Object Notation. It is a lightweight, text-based data format that is commonly used to exchange data between a server and a web application.

```
{
  "name": "Alice",
  "age": 30,
  "isStudent": false,
  "list of courses": ["Mathematics", "Physics", "Computer Science"]
}
```

- `JSON.stringify()`: This method is used to convert a JavaScript object into a JSON string. This is useful when you want to store or transmit data in a format that can be easily shared or transferred between systems.

```
const user = {
  name: "John",
  age: 30,
  isAdmin: true
};

const jsonString = JSON.stringify(user);
console.log(jsonString); // '{"name":"John","age":30,"isAdmin":true}'
```

- `JSON.parse()`: This method converts a JSON string back into a JavaScript object. This is useful when you retrieve JSON data from a web server or localStorage and you need to manipulate the data in your application.

```
const jsonString = '{"name":"John","age":30,"isAdmin":true}';
const userObject = JSON.parse(jsonString);

// result: { name: 'John', age: 30, isAdmin: true }
console.log(userObject);
```

Working with Loops

- **`for` Loop**: This type of loop is used to repeat a block of code a certain number of times. This loop is broken up into three parts: the initialization statement, the condition, and the increment/decrement statement. The initialization statement is executed before the loop starts. It is typically used to initialize a counter variable. The condition is evaluated before each iteration of the loop. An iteration is a single pass through the loop. If the condition is `true`, the code block inside the loop is executed. If the condition is `false`, the loop stops and you move on to the next block of code. The increment/decrement statement is executed after each iteration of the loop. It is typically used to increment or decrement the counter variable.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

- **`for...of` Loop**: This type of loop is used when you need to loop over values from an iterable. Examples of iterables are arrays and strings.

```
const numbers = [1, 2, 3, 4, 5];

for (const num of numbers) {
  console.log(num);
}
```

- **`for...in` Loop**: This type of loop is best used when you need to loop over the properties of an object. This loop will iterate over all enumerable properties of an object, including inherited properties and non-numeric properties.

```
const fruit = {
  name: 'apple',
  color: 'red',
  price: 0.99
};

for (const prop in fruit) {
  console.log(fruit[prop]);
}
```

- **`while` Loop**: This type of loop will run a block of code as long as the condition is `true`.

```
let i = 5;

while (i > 0) {
  console.log(i);
  i--;
}
```

- **`do...while` Loop**: This type of loop will execute the block of code at least once before checking the condition.

```
let userInput;

do {
  userInput = prompt("Please enter a number between 1 and 10");
} while (Number(userInput) < 1 || Number(userInput) > 10);

alert("You entered a valid number!");
```

`break` and `continue` Statements

- **Definition**: A `break` statement is used to exit a loop early, while a `continue` statement is used to skip the current iteration of a loop and move to the next one.

```
// Example of break statement
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}

// Output: 0, 1, 2, 3, and 4

// Example of continue statement
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    continue;
  }
  console.log(i);
}

// Output: 0, 1, 2, 3, 4, 6, 7, 8, and 9
```

String Constructor and `toString()` Method

- **Definition**: A string object is used to represent a sequence of characters. String objects are created using the `String` constructor function, which wraps the primitive value in an object.

```
const greetingObject = new String("Hello, world!");

console.log(typeof greetingObject); // "object"
```

- **`toString()` Method**: This method converts a value to its string representation. It is a method you can use for numbers, booleans, arrays, and objects.

```
const num = 10;
console.log(num.toString()); // "10"

const arr = [1, 2, 3];
console.log(arr.toString()); // "1,2,3"
```

This method accepts an optional radix which is a number from 2 to 36. This radix represents the base, such as base 2 for binary or base 8 for octal. If the radix is not specified, it defaults to base 10, which is decimal.

```
const num = 10;
console.log(num.toString(2)); // "1010"(binary)
```

Number Constructor

- **Definition**: The `Number` constructor is used to create a number object. The number object contains a few helpful properties and methods like the `isNaN` and `toFixed` method. Most of the time, you will be using the `Number` constructor to convert other data types to the number data type.

```
const myNum = new Number("34");
console.log(typeof myNum); // "object"

const num = Number('100');
console.log(num); // 100

console.log(typeof num); // number
```

Best Practices for Naming Variables and Functions

- **camelCasing**: By convention, JavaScript developers will use camel casing for naming variables and functions. Camel casing is where the first word is all lowercase and the following words start with a capital letter. Ex. `isLoading`.
- **Naming Booleans**: For boolean variables, it's a common practice to use prefixes such as "is", "has", or "can".

```
let isLoading = true;
let hasPermission = false;
let canEdit = true;
```

- **Naming Functions**: For functions, the name should clearly indicate what the function does. For functions that return a boolean (often called predicates), you can use the same "is", "has", or "can" prefixes. When you have functions that retrieve data, it is common to start with the word "get". When you have functions that set data, it is common to start with the word "set". For event handler functions, you might prefix with "handle" or suffix with "Handler".

```
function getUserData() { /* ... */ }

function isValidEmail(email) { /* ... */ }

function getProductDetails(productId) { /* ... */ }

function setUserPreferences(preferences) { /* ... */ }

function handleClick() { /* ... */ }
```

- **Naming Variables Inside Loops**: When naming iterator variables in loops, it's common to use single letters like `i`, `j`, or `k`.

```
for (let i = 0; i < array.length; i++) { /* ... */ }
```

Working with Sparse Arrays

- **Definition**: It is possible to have arrays with empty slots. Empty slots are defined as slots with nothing in them. This is different than array slots with the value of `undefined`. These types of arrays are known as sparse arrays.

```
const sparseArray = [1, , , 4];
console.log(sparseArray.length); // 4
```

Linters and Formatters

- **Linters**: A linter is a static code analysis tool that flags programming errors, bugs, stylistic errors, and suspicious constructs. An example of a common linter would be ESLint.
- **Formatters**: Formatters are tools that automatically format your code to adhere to a specific style guide. An example of a common formatter is Prettier.

Memory Management

- **Definition**: Memory management is the process of controlling the memory, allocating it when needed and freeing it up when it's no longer needed. JavaScript uses automatic memory management. This means that JavaScript (more specifically, the JavaScript engine in your web browser) takes care of memory allocation and deallocation for you. You don't have to explicitly free up memory in your code. This automatic process is often called "garbage collection."

Closures

- **Definition**: A closure is a function that has access to variables in its outer (enclosing) lexical scope, even after the outer function has returned.

```
function outerFunction(x) {
  let y = 10;
  function innerFunction() {
    console.log(x + y);
  }
  return innerFunction;
}

let closure = outerFunction(5);
closure(); // 15
```

`var` Keyword and Hoisting

- **Definition**: `var` was the original way to declare variables before 2015. But there were some issues that came with `var` in terms of scope, redeclaration and more. So that is why modern JavaScript programming uses `let` and `const` instead.
- **Redeclaring Variables with `var`**: If you try to redeclare a variable using `let`, then you would get a `SyntaxError`. But with `var`, you are allowed to redeclare a variable.

```
// Uncaught SyntaxError: Identifier 'num' has already been declared
let num = 19;
let num = 18;

var myNum = 5;
var myNum = 10; // This is allowed and doesn't throw an error
```

```
console.log(myNum) // 10
```

- **`var` and Scope**: Variables declared with `var` inside a block (like an `if` statement or a `for` loop) are still accessible outside that block.

```
if (true) {
  var num = 5;
}
console.log(num); // 5
```

- **Hoisting**: This is JavaScript's default behavior of moving declarations to the top of their respective scopes during the compilation phase before the code is executed. When you declare a variable using the `var` keyword, JavaScript hoists the declaration to the top of its scope.

```
console.log(num); // undefined
var num = 5;
console.log(num); // 5
```

When you declare a function using the function declaration syntax, both the function name and the function body are hoisted. This means you can call a function before you've declared it in your code.

```
sayHello(); // "Hello, World!"

function sayHello() {
  console.log("Hello, World!");
}
```

Variable declarations made with `let` or `const` are hoisted, but they are not initialized, and you can't access them before the actual declaration in your code. This behavior is often referred to as the "temporal dead zone".

```
console.log(num); // Throws a ReferenceError
let num = 10;
```

Working with Imports, Exports and Modules

- **Module**: This is a self-contained unit of code that encapsulates related functions, classes, or variables. To create a module, you write your JavaScript code in a separate file.
- **Exports**: Any variables, functions, or classes you want to make available to other parts of your application need to be explicitly exported using the `export` keyword. There are two types of export: named export and default export.
- **Imports**: To use the exported items in another part of your application, you need to import them using the `import` keyword. The types can be named import, default import, and namespace import.

```
// Within a file called math.js, we export the following functions:

// Named export
export function add(num1, num2) {
  return num1 + num2;
}
```

```
// Default export
export default function subtract(num1, num2) {
  return num1 - num2;
}

// Within another file, we can import the functions from math.js.

// Named import - This line imports the add function.
// The name of the function must exactly match the one exported from math.js.
import { add } from './math.js';

// Default import - This line imports the subtract function.
// The name of the function can be anything.
import subtractFunc from './math.js';

// Namespace import - This line imports everything from the file.
import * as Math from './math.js';

console.log(add(5, 3)); // 8
console.log(subtractFunc(5, 3)); // 2
console.log(Math.add(5, 3)); // 8
console.log(Math.subtract(5, 3)); // 2
```

Callback Functions and the `forEach` Method

- **Definition**: In JavaScript, a callback function is a function that is passed as an argument to another function and is executed after the main function has finished its execution.
- **`forEach()` Method**: This method is used to iterate over each element in an array and perform an operation on each element. The callback function in `forEach` can take up to three arguments: the current element, the index of the current element, and the array that `forEach` was called upon.

```
const numbers = [1, 2, 3, 4, 5];

// Result: 2 4 6 8 10
numbers.forEach((number) => {
  console.log(number * 2);
});
```

Higher Order Functions

- **Definition**: A higher-order function takes one or more functions for the arguments and returns a function or value for the result.

```
function operateOnArray(arr, operation) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(operation(arr[i]));
  }
  return result;
}

function double(x) {
  return x * 2;
}

const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = operateOnArray(numbers, double);
console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

- `map()` **Method**: This method is used to create a new array by applying a given function to each element of the original array. The callback function can accept up to three arguments: the current element, the index of the current element, and the array that `map` was called upon.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((num) => num * 2);

console.log(numbers); // [1, 2, 3, 4, 5]
console.log(doubled); // [2, 4, 6, 8, 10]
```

- `filter()` **Method**: This method is used to create a new array with elements that pass a specified test, making it useful for selectively extracting items based on criteria. Just like the `map` method, the callback function for the `filter` method accepts the same three arguments: the current element being processed, the index, and the array.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const evenNumbers = numbers.filter((num) => num % 2 === 0);

console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

- `reduce()` **Method**: This method is used to process an array and condense it into a single value. This single value can be a number, a string, an object, or even another array. The `reduce()` method works by applying a function to each element in the array, in order, passing the result of each calculation on to the next. This function is often called the reducer function. The reducer function takes two main parameters: an accumulator and the current value. The accumulator is where you store the running result of your operations, and the current value is the array element being processed.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
  0
);

console.log(sum); // 15
```

Method Chaining

- **Definition**: Method chaining is a programming technique that allows you to call multiple methods on the same object in a single line of code. This technique can make your code more readable and concise, especially when performing a series of operations on the same object.

```
const result = "  Hello, World!  "
  .trim()
  .toLowerCase()
  .replace("world", "JavaScript");

console.log(result); // "hello, JavaScript!"
```

Working with the `sort` Method

- **Definition**: The `sort` method is used to sort the elements of an array and return a reference to the sorted array. No copy is made in this case because the elements are sorted in place.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();

console.log(fruits); // ["Apple", "Banana", "Mango", "Orange"]
```

If you need to sort numbers, then you will need to pass in a compare function. The `sort` method converts the elements to strings and then compares their sequences of UTF-16 code units values. UTF-16 code units are the numeric values that represent the characters in the string. Examples of UTF-16 code units are the numbers 65, 66, and 67 which represent the characters "A", "B", and "C" respectively. So the number 200 appears before the number 3 in an array, because the string "200" comes before the string "3" when comparing their UTF-16 code units.

```
const numbers = [414, 200, 5, 10, 3];

numbers.sort((a, b) => a - b);

console.log(numbers); // [3, 5, 10, 200, 414]
```

The parameters `a` and `b` are the two elements being compared. The compare function should return a negative value if `a` should come before `b`, a positive value if `a` should come after `b`, and zero if `a` and `b` are equal.

Working with the `every` and `some` Methods

- **`every()` Method**: This method tests whether all elements in an array pass a test implemented by a provided function. The `every()` method returns `true` if the provided function returns `true` for all elements in the array. If any element fails the test, the method immediately returns `false` and stops checking the remaining elements.

```
const numbers = [2, 4, 6, 8, 10];
const hasAllEvenNumbers = numbers.every((num) => num % 2 === 0);

console.log(hasAllEvenNumbers); // true
```

- **`some()` Method**: This method checks if at least one element passes the test. The `some()` method returns `true` as soon as it finds an element that passes the test. If no elements pass the test, it returns `false`.

```
const numbers = [1, 3, 5, 7, 8, 9];
const hasSomeEvenNumbers = numbers.some((num) => num % 2 === 0);

console.log(hasSomeEvenNumbers); // true
```

Working with the DOM and Web APIs

- **API**: An API (Application Programming Interface) is a set of rules and protocols that allow software applications to communicate with each other and exchange data efficiently.
- **Web API**: Web APIs are specifically designed for web applications. These types of APIs are often divided into two main categories: browser APIs and third-party APIs.
- **Browser APIs**: These APIs expose data from the browser. As a web developer, you can access and manipulate this data using JavaScript.
- **Third-Party APIs**: These are not built into the browser by default. You have to retrieve their code in some way. Usually, they will have detailed documentation explaining how to use their services. An

example is the Google Maps API, which you can use to display interactive maps on your website.

- **DOM**: The DOM stands for Document Object Model. It's a programming interface that lets you interact with HTML documents. With the DOM, you can add, modify, or delete elements on a webpage. The root of the DOM tree is the `html` element. It's the top-level container for all the content of an HTML document. All other nodes are descendants of this root node. Then, below the root node, we find other nodes in the hierarchy. A parent node is an element that contains other elements. A child node is an element that is contained within another element.
- `navigator` **Interface**: This provides information about the browser environment, such as the user agent string, the platform, and the version of the browser. A user agent string is a text string that identifies the browser and operating system being used.
- `window` **Interface**: This represents the browser window that contains the DOM document. It provides methods and properties for interacting with the browser window, such as resizing the window, opening new windows, and navigating to different URLs.

Working with the `querySelector()`, `querySelectorAll()` and `getElementById()` Methods

- `getElementById()` **Method**: This method is used to get an object that represents the HTML element with the specified `id`. Remember that IDs must be unique in every HTML document, so this method will only return one Element object.

```
<div id="container"></div>
```

```
const container = document.getElementById("container");
```

- `querySelector()` **Method**: This method is used to get the first element in the HTML document that matches the CSS selector passed as an argument.

```
<section class="section"></section>
```

```
const section = document.querySelector(".section");
```

- `querySelectorAll()` **Method**: You can use this method to get a list of all the DOM elements that match a specific CSS selector.

```
<ul class="ingredients">
  <li>Sugar</li>
  <li>Milk</li>
  <li>Eggs</li>
</ul>
```

```
const ingredients = document.querySelectorAll('ul.ingredients li');
```

Working with the `innerText()`, `innerHTML()`, `createElement()` and `textContent()` Methods

- `innerHTML` **Property**: This is a property of the `Element` that is used to set or update parts of the HTML markup.

```
<div id="container">
  <!-- Add new elements here -->
</div>
```

```
const container = document.getElementById("container");
container.innerHTML = '<ul><li>Cheese</li><li>Tomato</li></ul>';
```

- `createElement` **Method**: This is used to create an HTML element.

```
const img = document.createElement("img");
```

- `innerText`: This represents the visible text content of the HTML element and its descendants.

```
<div id="container">
  <p>Hello, World!</p>
  <p>I'm learning JavaScript</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.innerText);
```

- `textContent`: This returns the plain text content of an element, including all the text within its descendants.

```
<div id="container">
  <p>Hello, World!</p>
  <p>I'm learning JavaScript</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.textContent);
```

Working with the `appendChild()` and `removeChild()` Methods

- `appendChild()` **Method**: This method is used to add a node to the end of the list of children of a specified parent node.

```
<ul id="desserts">
  <li>Cake</li>
  <li>Pie</li>
</ul>
```

```
const dessertsList = document.getElementById("desserts");
const listItem = document.createElement("li");
```

```
listItem.textContent = "Cookies";
dessertsList.appendChild(listItem);
```

- `removeChild()` **Method**: This method is used to remove a node from the DOM.

```
<section id="example-section">
  <h2>Example sub heading</h2>
  <p>first paragraph</p>
  <p>second paragraph</p>
</section>
```

```
const sectionEl = document.getElementById("example-section");
const lastParagraph = document.querySelector("#example-section p:last-of-
type");

sectionEl.removeChild(lastParagraph);
```

Work with the `setAttribute` Method

- **Definition**: This method is used to set the attribute for a given element. If the attribute already exists, then the value is updated. Otherwise, a new attribute is added with a value.

```
<p id="para">I am a paragraph</p>
```

```
const para = document.getElementById("para");
para.setAttribute("class", "my-class");
```

Event Object

- **Definition**: The `Event` object is a payload that triggers when a user interacts with your web page in some way. These interactions can be anything from clicking on a button or focusing an input to shaking their mobile device. All `Event` objects will have the `type` property. This property reveals the type of event that triggered the payload, such as keydown or click. These values will correspond to the same values you might pass to `addEventListener`(), where you can capture and utilize the `Event` object.

`addEventListener()` and `removeEventListener()` Methods

- `addEventListener` **Method**: This method is used to listen for events. It takes two arguments: the event you want to listen for and a function that will be called when the event occurs. Some common examples of events would be click events, input events, and change events.

```
const btn = document.getElementById("btn");

btn.addEventListener("click", () => alert("You clicked the button"));
```

- `removeEventListener` **Method**: This method is used to remove an event listener that was previously added to an element using the `addEventListener` method. This is useful when you want to stop listening for a particular event on an element.

```
```

```
const bodyEl = document.querySelector("body");
const para = document.getElementById("para");
const btn = document.getElementById("btn");

let isBgColorGrey = true;

function toggleBgColor() {
  bodyEl.style.backgroundColor = isBgColorGrey ? "blue" : "grey";
  isBgColorGrey = !isBgColorGrey;
}

btn.addEventListener("click", toggleBgColor);

para.addEventListener("mouseover", () => {
  btn.removeEventListener("click", toggleBgColor);
});
```

- **Inline Event Handlers**: Inline event handlers are special attributes on an HTML element that are used to execute JavaScript code when an event occurs. In modern JavaScript, inline event handlers are not considered best practice. It is preferred to use the `addEventListener` method instead.

```
<button onclick="alert('Hello World!')">Show alert</button>
```

DOMContentLoaded

- **Definition**: The `DOMContentLoaded` event is fired when everything in the HTML document has been loaded and parsed. If you have external stylesheets or images, the `DOMContentLoaded` event will not wait for those to be loaded. It will only wait for the HTML to be loaded.

Working with `style` and `classList`

- **`Element.style` Property**: This property is a read-only property that represents the inline style of an element. You can use this property to get or set the style of an element.

```
const paraEl = document.getElementById("para");
paraEl.style.color = "red";
```

- **`Element.classList` Property**: This property is a read-only property that can be used to add, remove, or toggle classes on an element.

```
// Example adding a class
const paraEl = document.getElementById("para");
paraEl.classList.add("highlight");

// Example removing a class
paraEl.classList.remove("blue-background");

// Example toggling a class
const menu = document.getElementById("menu");
const toggleBtn = document.getElementById("toggle-btn");

toggleBtn.addEventListener("click", () => menu.classList.toggle("show"));
```

Working with the `setTimeout` and `setInterval` Methods

- `setTimeout()` **Method**: This method lets you delay an action for a specified time.

```
setTimeout(() => {
 console.log('This runs after 3 seconds');
}, 3000);
```

- `setInterval()` **Method**: This method keeps running a piece of code repeatedly at a set interval. Since `setInterval()` keeps executing the provided function at the specified interval, you might want to stop it. For this, you have to use the `clearInterval()` method.

```
setInterval(() => {
 console.log('This runs every 2 seconds');
}, 2000);

// Example using clearInterval
const intervalID = setInterval(() => {
 console.log('This will stop after 5 seconds');
}, 1000);

setTimeout(() => {
 clearInterval(intervalID);
}, 5000);
```

The `requestAnimationFrame()` Method

- **Definition**: This method allows you to schedule the next step of your animation before the next screen repaint, resulting in a fluid and visually appealing experience. The next screen repaint refers to the moment when the browser refreshes the visual display of the web page. This happens multiple times per second, typically around 60 times (or 60 frames per second) on most displays.

```
function animate() {
 // Update the animation...
 // for example, move an element, change a style, and more.
 update();
 // Request the next frame
 requestAnimationFrame(animate);
}
```

Web Animations API

- **Definition**: The Web Animations API lets you create and control animations directly inside JavaScript.

```
const square = document.querySelector('#square');

const animation = square.animate(
 [{ transform: 'translateX(0px)' }, { transform: 'translateX(100px)' }],
 {
   duration: 2000, // makes animation lasts 2 seconds
   iterations: Infinity, // loops indefinitely
   direction: 'alternate', // moves back and forth
   easing: 'ease-in-out', // smooth easing
 }
);
```

The Canvas API

- **Definition**: The Canvas API is a powerful tool that lets you manipulate graphics right inside your JavaScript file. To work with the Canvas API, you first need to provide a `canvas` element in HTML. This element acts as a drawing surface you can manipulate with the instance methods and properties of the interfaces in the Canvas API. This API has interfaces like `HTMLCanvasElement`, `CanvasRenderingContext2D`, `CanvasGradient`, `CanvasPattern`, and `TextMetrics` which contain methods and properties you can use to create graphics in your JavaScript file.

```
<canvas id="my-canvas" width="400" height="400"></canvas>
```

```
const canvas = document.getElementById('my-canvas');

// Access the drawing context of the canvas.
// "2d" allows you to draw in two dimensions
const ctx = canvas.getContext('2d');

// Set the background color
ctx.fillStyle = 'crimson';

// Draw a rectangle
ctx.fillRect(1, 1, 150, 100);
```

Opening and Closing Dialogs and Modals with JavaScript

- **Modal and Dialog Definitions**: Dialogs let you display important information or actions to users. With the HTML built-in dialog element, you can easily create these dialogs (both modal and non-modal dialogs) in your web apps. A modal dialog is a type of dialog that forces the user to interact with it before they can access the rest of the application or webpage. In contrast, a non-modal dialog allows the user to continue interacting with other parts of the page or application even when the dialog is open. It doesn't prevent access to the rest of the content.
- `showModal()` **Method**: This method is used to open a modal.

```
<dialog id="my-modal">
   <p>This is a modal dialog.</p>
</dialog>
<button id="open-modal">Open Modal Dialog</button>
```

```
const dialog = document.getElementById('my-modal');
const openButton = document.getElementById('open-modal');

openButton.addEventListener('click', () => {
  dialog.showModal();
});
```

- `close()` **Method**: This method is used to close the modal.

```
<dialog id="my-modal">
   <p>This is a modal dialog.</p>
   <button id="close-modal">Close Modal</button>
</dialog>
<button id="open-modal">Open Modal Dialog</button>
```

```
const dialog = document.getElementById('my-modal');
const openButton = document.getElementById('open-modal');
const closeButton = document.getElementById('close-modal');

openButton.addEventListener('click', () => {
  dialog.show();
});

closeButton.addEventListener('click', () => {
  dialog.close();
});
```

The Change Event

- **Definition**: The change event is a special event which is fired when the user modifies the value of certain input elements. Examples would include when a checkbox or a radio button is ticked. Or when the user makes a selection from something like a date picker or dropdown menu.

```
<label>
  Choose a programming language:
  <select class="language" name="language">
    <option value="">---Select One---</option>
    <option value="JavaScript">JavaScript</option>
    <option value="Python">Python</option>
    <option value="C++">C++</option>
  </select>
</label>

<p class="result"></p>
```

```
const selectEl = document.querySelector(".language");
const result = document.querySelector(".result");

selectEl.addEventListener("change", (e) => {
  result.textContent = `You enjoy programming in ${e.target.value}.`;
});
```

Event Bubbling

- **Definition**: Event bubbling, or propagation, refers to how an event "bubbles up" to parent objects when triggered.

Event Delegation

- **Definition**: Event delegation is the process of listening to events that have bubbled up to a parent, rather than handling them directly on the element that triggered them.

JavaScript and Accessibility

Common ARIA Accessibility Attributes

- `aria-expanded` **attribute**: Used to convey the state of a toggle (or disclosure) feature to screen reader users.
- `aria-haspopup` **attribute**: This state is used to indicate that an interactive element will trigger a pop-up element when activated. You can only use the `aria-haspopup` attribute when the pop-up has one of the following roles: `menu`, `listbox`, `tree`, `grid`, or `dialog`. The value of `aria-haspopup` must be either one of these roles or `true`, which is the same as `menu`.

- `aria-checked` **attribute**: This attribute is used to indicate whether an element is in the checked state. It is most commonly used when creating custom checkboxes, radio buttons, switches, and listboxes.
- `aria-disabled` **attribute**: This state is used to indicate that an element is disabled only to people using assistive technologies, such as screen readers.
- `aria-selected` **attribute**: This state is used to indicate that an element is selected. You can use this state on custom controls like a tabbed interface, a listbox, or a grid.
- `aria-controls` **attribute**: Used to associate an element with another element that it controls. This helps people using assistive technologies understand the relationship between the elements.
- `hidden` **attribute**: Hides inactive panels from both visual and assistive technology users.

Working with Live Regions and Dynamic Content

- `aria-live` **attribute**: Makes part of a webpage a live region, meaning any updates inside that area will be announced by a screen reader so users don't miss important changes.
- `polite` **value**: Most live regions use this value. This value means that the update is not urgent, so the screen reader can wait until it finishes any current announcement or the user completes their current action before announcing the update.

Here is an example of a live region that is dynamically updated by JavaScript:

```
<div aria-live="polite" id="status"></div>
```

```
const statusEl = document.getElementById("status");
statusEl.textContent = "Your file has been successfully uploaded.";
```

- `contenteditable` **attribute**: Turns the element into a live editor, allowing users to update its content as if it were a text field. When there is no visible label or heading for a contenteditable region, add an accessible name using the `aria-label` attribute to help screen reader users understand the purpose of the editable area.

```
<div contenteditable="true" aria-label="Note editor">
  Editable content goes here
</div>
```

`focus` and `blur` Events

- `blur` **event**: Fires when an element loses focus.

```
element.addEventListener("blur", () => {
  // Handle when user leaves the element
});
```

- `focus` **event**: Fires when an element receives focus.

```
element.addEventListener("focus", () => {
  // Handle when user enters the element
});
```

Common Types of Error Messages

- **SyntaxError**: These errors happen when you write something incorrectly in your code, like missing a parenthesis, or a bracket. Think of it like a grammar mistake in a sentence.

```
const arr = ["Beau", "Quincy" "Tom"]
```

- **ReferenceError**: There are several types of Reference Errors, triggered in different ways. The first type of reference error would be not defined variables. Another example of a ReferenceError is trying to access a variable, declared with `let` or `const`, before it has been defined.

```
console.log(num);
const num = 50;
```

- **TypeError**: These errors occur when you try to perform an operation on the wrong type.

```
const developerObj = {
  name: "Jessica",
  country: "USA",
  isEmployed: true
};

developerObj.map()
```

- **RangeError**: These errors happen when your code tries to use a value that's outside the range of what JavaScript can handle.

```
const arr = [];
arr.length = -1;
```

The `throw` Statement

- **Definition**: The `throw` statement in JavaScript is used to throw a user-defined exception. An exception in programming, is when an unexpected event happens and disrupts the normal flow of the program.

```
function validateNumber(input) {
  if (typeof input !== "number") {
    throw new TypeError("Expected a number, but received " + typeof input);
  }
  return input * 2;
}
```

`try...catch...finally`

- **Definition**: The `try` block is used to wrap code that might throw an error. It acts as a safe space to try something that could fail. The `catch` block captures and handles errors that occur in the try block. You can use the error object inside catch to inspect what went wrong. The `finally` block runs after the try and catch blocks, regardless of whether an error occurred. It's commonly used for cleanup tasks, such as closing files or releasing resources.

```
function processInput(input) {
  if (typeof input !== "string") {
```

```
    throw new TypeError("Input must be a string.");
  }

  return input.toUpperCase();
}

try {
  console.log("Starting to process input...");
  const result = processInput(9);
  console.log("Processed result:", result);
} catch (error) {
  console.error("Error occurred:", error.message);
}
```

Debugging Techniques

- `debugger` **Statement**: This statement lets you pause your code at a specific line to investigate what's going on in the program.

```
let firstNumber = 5;
let secondNumber = 10;

debugger; // Code execution pauses here
let sum = firstNumber + secondNumber;

console.log(sum);
```

- **Breakpoints**: Breakpoints let you pause the execution of your code at a specific line of your choice. After the pause, you can inspect variables, evaluate expressions, and examine the call stack.
- **Watchers**: Watch expressions lets you monitor the values of variables or expressions as the code runs even if they are out of the current scope.
- **Profiling**: Profiling helps you identify performance bottlenecks by letting you capture screenshots and record CPU usage, function calls, and execution time.
- `console.dir()`: This method is used to display an interactive list of the properties of a specified JavaScript object. It outputs a hierarchical listing that can be expanded to see all nested properties.

```
console.dir(document);
```

- `console.table()`: This method displays tabular data as a table in the console. It takes one mandatory argument, which must be an array or an object, and one optional argument to specify which properties (columns) to display.

Regular Expressions and Common Methods

- **Definition**: Regular Expressions, or Regex, are used to create a "pattern", which you can then use to check against a string, extract text, and more.

```
const regex = /freeCodeCamp/;
```

- `test()` **Method**: This method accepts a string, which is the string to test for matches against the regular expression. This method will return a boolean if the string matches the regex.

```
const regex = /freeCodeCamp/;
const test = regex.test("e");
```

```
console.log(test); // false
```

- `match()` **Method**: This method accepts a regular expression, although you can also pass a string which will be constructed into a regular expression. The `match` method returns the match array for the string.

```
const regex = /freeCodeCamp/;
const match = "freeCodeCamp".match(regex);
console.log(match); // ["freeCodeCamp"]
```

- `replace()` **Method**: This method accepts two arguments: the regular expression to match (or a string), and the string to replace the match with (or a function to run against each match).

```
const regex = /Jessica/;
const str = "Jessica is rly kewl";
const replaced = str.replace(regex, "freeCodeCamp");
console.log(replaced); // "freeCodeCamp is rly kewl"
```

- `replaceAll` **Method**: This method is used to replace all occurrences of a specified pattern with a new string. This method will throw an error if you give it a regular expression without the global modifier.

```
const text = "I hate JavaScript! I hate programming!";
const newText = text.replaceAll("hate", "love");
console.log(newText);  // "I love JavaScript! I love programming!"
```

- `matchAll` **Method**: This method is used to retrieve all matches of a given regular expression in a string, including capturing groups, and returns them as an iterator. An iterator is an object that allows you to go through (or "iterate over") a collection of items.

```
const str = "JavaScript, Python, JavaScript, Swift, JavaScript";
const regex = /JavaScript/g;

const iterator = str.matchAll(regex);

for (let match of iterator) {
  console.log(match[0]); // "JavaScript" for each match
}
```

Regular Expression Modifiers

- **Definition**: Modifiers, often referred to as "flags", modify the behavior of a regular expression.
- `i` **Flag**: This flag makes a regex ignore case.

```
const regex = /freeCodeCamp/i;
console.log(regex.test("freecodecamp")); // true
console.log(regex.test("FREECODECAMP")); // true
```

- `g` **Flag**: This flag, or global modifier, allows your regular expression to match a pattern more than once.

```
const regex = /freeCodeCamp/gi;
console.log(regex.test("freeCodeCamp")); // true
console.log(regex.test("freeCodeCamp is great")); // false
```

- **Anchor Definition**: The `^` anchor, at the beginning of the regular expression, says "match the start of the string". The `$` anchor, at the end of the regular expression, says "match the end of the string".

```
const start = /^freeCodeCamp/i;
const end = /freeCodeCamp$/i;
console.log(start.test("freecodecamp")); // true
console.log(end.test("freecodecamp")); // true
```

- **`m` Flag**: Anchors look for the beginning and end of the entire string. But you can make a regex handle multiple lines with the `m` flag, or the multi-line modifier.flag, or the multi-line modifier.

```
const start = /^freecodecamp/im;
const end = /freecodecamp$/im;
const str = `I love
freecodecamp
it's my favorite
`;
console.log(start.test(str)); // true
console.log(end.test(str)); // true
```

- **`d` Flag**: This flag expands the information you get in a match object.

```
const regex = /freecodecamp/di;
const string = "we love freecodecamp isn't freecodecamp great?";
console.log(string.match(regex));
```

- **`u` Flag**: This expands the functionality of a regular expression to allow it to match special unicode characters. The `u` flag gives you access to special classes like the `Extended_Pictographic` to match most emoji. There is also a `v` flag, which further expands the functionality of the unicode matching.
- **`y` Flag**: The sticky modifier behaves very similarly to the global modifier, but with a few exceptions. The biggest one is that a global regular expression will start from lastIndex and search the entire remainder of the string for another match, but a sticky regular expression will return null and reset the lastIndex to 0 if there is not immediately a match at the previous lastIndex.
- **`s` Flag**: The single-line modifier allows a wildcard character, represented by a `.` in regex, to match linebreaks - effectively treating the string as a single line of text.

Character Classes

- **Wildcard `.`**: Character classes are a special syntax you can use to match sets or subsets of characters. The first character class you should learn is the wild card class. The wild card is represented by a period, or dot, and matches ANY single character EXCEPT line breaks. To allow the wildcard class to match line breaks, remember that you would need to use the `s` flag.

```
const regex = /a./;
```

- **`\d`**: This will match all digits (`0-9`) in a string.

```
const regex = /\d/;
```

- `\w`: This is used to match any word character (`a-z0-9_`) in a string. A word character is defined as any letter, from a to z, or a number from 0 to 9, or the underscore character.

```
const regex = /\w/;
```

- `\s`: The white-space class `\s`, represented by a backslash followed by an `s`. This character class will match any white space, including new lines, spaces, tabs, and special unicode space characters.
- **Negating Special Character Classes**: To negate one of these character classes, instead of using a lowercase letter after the backslash, you can use the uppercase equivalent. The following example does not match a numerical character. Instead, it matches any single character that is NOT a numerical character.

```
const regex = /\D/;
```

- **Custom Character Classes**: You can create custom character classes by placing the character you wish match inside a set of square brackets.

```
const regex = /[abcdf]/;
```

Lookahead and Lookbehind Assertions

- **Definition**: Lookahead and lookbehind assertions allow you to match specific patterns based on the presence or lack of surrounding patterns.
- **Positive Lookahead Assertion**: This assertion will match a pattern when the pattern is followed by another pattern. To construct a positive lookahead, you need to start with the pattern you want to match. Then, use parentheses to wrap the pattern you want to use as your condition. After the opening parenthesis, use `?=` to define that pattern as a positive lookahead.

```
const regex = /free(?=code)/i;
```

- **Negative Lookahead Assertion**: This is a type of condition used in regular expressions to check that a certain pattern does not occur ahead in the string.

```
const regex = /free(?!code)/i;
```

- **Positive Lookbehind Assertion**: This assertion will match a pattern only if it is preceded by another specific pattern, without including the preceding pattern in the match.

```
const regex = /(?<=free)code/i;
```

- **Negative Lookbehind Assertion**: This assertion ensures that a pattern is not preceded by another specific pattern. It matches only if the specified pattern is not immediately preceded by the given sequence, without including the preceding sequence in the match.

```
```

```
const regex = /(?<!free)code/i;
```

Regex Quantifiers

- **Definition**: Quantifiers in regular expressions specify how many times a pattern (or part of a pattern) should appear. They help control the number of occurrences of characters or groups in a match. The following example is used to match the previous character exactly four times.

```
const regex = /^\d{4}$/;
```

- `*` : Matches 0 or more occurrences of the preceding element.
- `+`: Matches 1 or more occurrences of the preceding element.
- `?`: Matches 0 or 1 occurrence of the preceding element.
- `{n}`: Matches exactly n occurrences of the preceding element.
- `{n,}`: Matches n or more occurrences of the preceding element.
- `{n,m}`: Matches between n and m occurrences of the preceding element.

Capturing Groups and Backreferences

- **Capturing Groups**: A capturing group allows you to "capture" a portion of the matched string to use however you might need. Capturing groups are defined by parentheses containing the pattern to capture, with no leading characters like a lookahead.

```
const regex = /free(code)camp/i;
```

- **Backreferences**: A backreference in regular expressions refers to a way to reuse a part of the pattern that was matched earlier in the same expression. It allows you to refer to a captured group (a part of the pattern in parentheses) by its number. For example, `$1` refers to the first captured group.

```
const regex = /free(co+de)camp/i;
console.log("freecooooooooodecamp".replace(regex, "paid$1world"));
```

Validating Forms with JavaScript

- **Constraint Validation API**: Certain HTML elements, such as the `textarea` and `input` elements, expose a constraint validation API. This API allows you to assert that the user's provided value for that element passes any HTML-level validation you have written, such as minimum length or pattern matching.
- `checkValidity()` **method**: This method returns `true` if the element matches all HTML validation (based on its attributes), and `false` if it fails.

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  if (!e.target.checkValidity()) {
    e.target.setCustomValidity("You must use a .com email.")
  }
})
```

- `reportValidity()` **Method**: This method tells the browser that the `input` is invalid.

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  if (!e.target.checkValidity()) {
    e.target.reportValidity();
  }
})
```

- **`validity` Property**: This property is used to get or set the validity state of form controls (like `<input>`, `<select>`, etc.) and provides information about whether the user input meets the constraints defined for that element (e.g., `required` fields, pattern constraints, maximum length, etc.).

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  console.log(e.target.validity);
})
```

- **`patternMismatch` Property**: This will be `true` if the value doesn't match the specified regular expression pattern.

`preventDefault()` Method

- **Definition**: Every event that triggers in the DOM has some sort of default behavior. The click event on a checkbox toggles the state of that checkbox, by default. Pressing the space bar on a focused button activates the button. The `preventDefault()` method on these `Event` objects stops that behavior from happening.

```
button.addEventListener('click', (event) => {
  // Prevent the default button click behavior
  event.preventDefault();
  alert('Button click prevented!');
});
```

Submitting Forms

- **Definition**: There are three ways a form can be submitted. The first is when the user clicks a button in the form which has the `type` attribute set to `submit`. The second is when the user presses the `Enter` key on any editable `input` field in the form. The third is through a JavaScript call to the `requestSubmit()` or `submit()` methods of the `form` element.
- **`action` Attribute**: The `action` attribute should contain either a URL or a relative path for the current domain. This value determines where the form attempts to send data - if you do not set an `action` attribute, the form will send data to the current page's URL.

```
<form action="https://freecodecamp.org">
  <input
    type="number"
    id="input"
    placeholder="Enter a number"
    name="number"
  />
  <button type="submit">Submit</button>
</form>
```

- `method` **Attribute**: This attribute accepts a standard `HTTP` method, such as `GET` or `POST`, and uses that method when making the request to the action URL. When a method is not set, the form will default to a `GET` request. The data in the form will be URL encoded as `name=value` pairs and appended to the action URL as query parameters.

```
<form action="/data" method="POST">
  <input
    type="number"
    id="input"
    placeholder="Enter a number"
    name="number"
  />
  <button type="submit">Submit</button>
</form>
```

- `enctype` **Attribute**: The `form` element accepts an `enctype` attribute, which represents the encoding type to use for the data. This attribute only accepts three values: `application/x-www-form-urlencoded` (which is the default, sending the data as a URL-encoded form body), `text/plain` (which sends the data in plaintext form, in `name=value` pairs separated by new lines), or `multipart/form-data`, which is specifically for handling forms with a file upload.

The `date()` Object and Common Methods

- **Definition**: The `date()` object is used to create, manipulate, and format dates and times in JavaScript. In the following example, the `new` keyword is used to create a new instance of the `Date` object, and the `Date` object is then assigned to the variable `now`. If you were to log the value of `now` to the console, you would see the current date and time based on the system clock of the computer running the code.

```
const now = new Date();
```

- `Date.now()` **Method**: This method is used to get the current date and time. `Date.now()` returns the number of milliseconds since January 1, 1970, 00:00:00 UTC. This is known as the Unix epoch time. Unix epoch time is a common way to represent dates and times in computer systems because it is an integer that can be easily stored and manipulated. UTC stands for Universal Time Coordinated, which is the primary time standard by which the world regulates clocks and time.
- `getDate()` **Method**: This method is used to get a day of the month based on the current date. `getDate()` will return an integer value between 1 and 31, depending on the day of the month. If the date is invalid, it will return `NaN` (Not a Number).

```
const now = new Date();
const date = now.getDate();
console.log(date); // 15
```

- `getMonth()` **Method**: This method is used to get the month. The month is zero-based, so January is 0, February is 1, and so on. In this example, the output is 2, which corresponds to March. If the month is invalid, it will return `NaN`.

```
const now = new Date();
const month = now.getMonth();
console.log(month); // 2
```

- `getFullYear()` **Method**: This method is used to get the full year. If the year is invalid, it will return `NaN`.

```
const now = new Date();
const year = now.getFullYear();
console.log(year); // 2024
```

Different Ways to Format Dates

- `toISOString()` **Method**: This method is used to format the date in an extended `ISO` (ISO 8601) format. ISO 8601 is an international standard for representing dates and times. The format is `YYYY-MM-DDTHH:mm:ss.sssZ`.

```
const date = new Date();
console.log(date.toISOString());
```

- `toLocaleDateString()` **Method**: This method is used to format the date based on the user's locale.

```
const date = new Date();
console.log(date.toLocaleDateString());  // 11/23/2024
```

The `toLocaleDateString()` method accepts two optional parameters: locales and options.

The locales parameter is a string representing the locale to use. For example, you can pass in `"en-US"` for English (United States) or `"fr-FR"` for French (France). If you don't pass in a locales parameter, the default locale is used. The second optional parameter is the options parameter. This parameter is an object that allows you to specify the format of the date string.

```
const date = new Date();
const options = {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric",
};
console.log(date.toLocaleDateString("en-GB", options)); // Saturday, November
23, 2024
```

`Audio` Constructor and Common Methods

- **Definition**: The `Audio` constructor, like other constructors, is a special function called with the `new` keyword. It returns an `HTMLAudioElement`, which you can then use to play audio for the user, or append to the DOM for the user to control themselves. When you call the constructor, you can optionally pass a URL as the (only) argument. This URL should point to the source of the audio file you want to play. Or, if you need to change the source dynamically, you can assign the URL to the `src` property of the returned audio element.
- `play()` **Method**: This method is used with the `audio` or `video` elements to begin playback for the media.

```
const audio = document.getElementById('audio');
```

```
// Starts playing the audio
audio.play();
```

- `pause()` **Method**: This method is used with the `audio` or `video` elements to pause playback for the media.

```
function pauseAudio() {
  const audio = document.getElementById('myAudio');
  audio.pause();  // Pauses the audio playback
}
```

- `addTextTrack()` **Method**: This method allows you to specify a text track to associate with the media element - which is especially helpful for adding subtitles to a video.
- `fastSeek()` **Method**: This method allows you to move the playback position to a specific time within the media.

Different Audio and Video Formats

- **MIME type**: A MIME type, standing for Multipurpose Internet Mail Extensions, is a standardized way to programmatically indicate a file type. The MIME type can tell an application, such as your browser, how to handle a specific file. In the case of audio and video, the MIME type indicates it is a multimedia format that can be embedded in the web page.
- `source` **Element**: This is used to specify a file type and source - and can include multiple different types by using multiple source elements. When you do this, the browser will determine the best format to use for the user's current environment.
- **MP3**: This is a type of digital file format used to store music, audio, or sound. It's a compressed version of a sound recording that makes the file size smaller, so it's easier to store and share. An MP3 file has the MIME type audio/mp3
- **MP4**: An MP4 is a type of digital file format used to store video and audio. It serves as a container that holds both the video (images) and the sound (music or speech) in one file. An MP4, can have the MIME type audio/mp4 OR video/mp4, depending on whether it's a video file or audio-only.

codecs

- **Definition**: A codec, short for "encoder/decoder", is an algorithm or software that can convert audio and video between analogue and digital formats. Codecs can be specified as part of the MIME type. The basic syntax to define a codec is to add a semi-colon after the media type, then `codecs=` and the codec.

HTMLMediaElement API

- **Definition**: The `HTMLMediaElement` API is used to control the behavior of audio and video elements on your page. It extends the base `HTMLElement` interface, so you have access to the base properties as well as these helpful methods. Examples of these methods include `play()`, `fastSeek()`, and `pause()`.

Media Capture and Streams API

- **Definition**: The Media Capture and Streams API, or the MediaStream API, is used to capture audio and video from your device. In order to use the API, you need to create the `MediaStream` object. You could do this with the constructor, but it would not be tied to the user's hardware. Instead, the `mediaDevices` property of the `global` navigator object has a `getUserMedia()` method for you to use.

```
window.navigator.mediaDevices.getUserMedia({
  audio: true,
```

```
    video: {
      width: {
        min: 1280,
        ideal: 1920,
        max: 3840
      },
      height: {
        min: 720,
        ideal: 1080,
        max: 2160
      }
    }
  });
```

Screen Capture API

- **Definition**: The Screen Capture API is used to record a user's screen. This API is exposed by calling the `getDisplayMedia()` method of the `mediaDevices` object and consuming the returned media stream.

MediaStream Recording API

- **Definition**: The MediaStream Recording API works in tandem with the MediaStreams APIs, allowing you to record a MediaStream (or even an `HTMLMediaElement` directly).

Media Source Extensions API

- **topic**: The Media Source Extensions API is what allows you to directly pass a user's webcam feed to a video element with the `srcObject` property.

Web Audio API

- **Definition**: The Web Audio API which powers everything audible on the web. This API includes important objects like an `AudioBuffer` (representing a Buffer specifically containing audio data) or the `AudioContext`.

Sets in JavaScript

- A `Set` is a built-in option for managing data collection.
- Sets ensure that each value in it appears only once, making it useful for eliminating duplicates from an array or handling collections of distinct values.
- You can create a `Set` using the `Set()` constructor:

```
const set = new Set([1, 2, 3, 4, 5]);
console.log(set); // Set { 1, 2, 3, 4, 5 }
```

- Sets can be manipulated using these methods:
    - `add()`: Adds a new element to the `Set`.
    - `delete()`: Removes an element from the `Set`.
    - `has()`: Checks if an element exists in the `Set`.
    - `clear()`: Removes all elements from the `Set`.

Weaksets in JavaScript

- `WeakSet` is a collection of objects that allows you to store weakly held objects.

Sets vs WeakSets

- Unlike Sets, a `WeakSet` does not support primitives like numbers or strings.
- A `WeakSet` only stores objects, and the references to those objects are "weak," meaning that if the object is not being used anywhere else in your code, it is removed automatically to free up memory.

Maps in JavaScript

- A `Map` is a built-in object that holds key-value pairs just like an object.
- Maps differ from the standard JavaScript objects with their ability to allow keys of any type, including objects, and functions.
- A `Map` provides better performance over the standard object when it comes to frequent addition and removals of key-value pairs.
- You can create a `Map` using the `Map()` constructor:

```
const map = new Map([
  ['flower', 'rose'],
  ['fruit', 'apple'],
  ['vegetable', 'carrot']
]);
console.log(map); // Map(3) { 'flower' => 'rose', 'fruit' => 'apple',
'vegetable' => 'carrot' }
```

- Maps can be manipulated using these methods:
    - `set()`: Adds a new key-value pair to the `Map`.
    - `get()`: Retrieves the value of a key from the `Map`.
    - `delete()`: Removes a key-value pair from the `Map`.
    - `has()`: Checks if a key exists in the `Map`.
    - `clear()`: Removes all key-value pairs from the `Map`.

WeakMaps in JavaScript

- A `WeakMap` is a collection of key-value pairs just like `Map`, but with weak references to the keys. The keys must be an object and the values can be anything you like.

Maps vs WeakMaps

- WeakMaps are similar to WeakSets in that they only store objects and the references to those objects are "weak."

Persistent Storage

- **Definition**: Persistent storage refers to a way of saving data in a way that it stays available even after the power is turned off or the device is restarted.

Create, Read, Update, Delete (CRUD)

- **Create**: This refers to the process of creating new data. For example, in a web app, this could be when a user adds a new post to a blog.
- **Read**: This is the operation where data is retrieved from a database. For instance, when you visit a blog post or view your profile on a website, you're performing a read operation to fetch and display data stored in the database.
- **Update**: This involves modifying existing data in the database. An example would be editing a blog post or updating your profile information.
- **Delete**: This is the operation that removes data from a database. For instance, when you delete a blog post or account, you're performing a delete operation.

HTTP Methods

- **Definition**: HTTP stands for Hypertext Transfer Protocol and it is the foundation for data communication on the web. There are HTTP methods which define the actions that can be performed on resources over the web. The common methods are GET, POST, PUT, PATCH, DELETE.
- `GET` **Method**: This is used to fetch data from a server.
- `POST` **Method**: This is used to submit data to a server which creates a new resource.
- `PUT` **Method**: This is used to update a resource by replacing it entirely.
- `PATCH` **Method**: This is used to partially update a resource.
- `DELETE` **Method**: This is used to remove records from a database.

`localStorage` and `sessionStorage` Properties

- **Web Storage API**: This API provides a mechanism for browsers to store key-value pairs right within the browser, allowing developers to store information that can be used across different page reloads and sessions. The two main components for the Web Storage API are the `localStorage` and `sessionStorage` properties.
- `localStorage` **Property**: `localStorage` is the part of the Web Storage API that allows data to persist even after the browser window is closed or the page is refreshed. This data remains available until it is explicitly removed by the application or the user.
- `localStorage.setItem()` **Method**: This method is used to store a key-value pair in `localStorage`.

```
localStorage.setItem('username', 'Jessica');
```

- `localStorage.getItem()` **Method**: This method is used to retrieve the value of a given key from `localStorage`.

```
localStorage.setItem('username', 'codingRules');

let username = localStorage.getItem('username');
console.log(username); // codingRules
```

- `localStorage.removeItem()` **Method**: This method is used to remove a specific item from `localStorage` using its key.

```
localStorage.removeItem('username');
```

- `localStorage.clear()` **Method**: This method is used to clear all of the stored data in `localStorage`.

```
localStorage.clear();
```

- `sessionStorage` **Property**: Stores data that lasts only for the current session and is cleared when the browser tab or window is closed.
- `sessionStorage.setItem()` **Method**: This method is used to store a key-value pair in `sessionStorage`.

```
sessionStorage.setItem('cart', '3 items');
```

- `sessionStorage.getItem()` **Method**: This method is used to retrieve the value of a given key from `sessionStorage`.

```
sessionStorage.setItem('cart', '3 items');

let cart = sessionStorage.getItem('cart');
console.log(cart); // '3 items'
```

- **`sessionStorage.removeItem()` Method**: This method is used to remove a specific item from `sessionStorage` using its key.

```
sessionStorage.removeItem('cart');
```

- **`sessionStorage.clear()` Method**: This method is used to clear all data stored in `sessionStorage`.

```
sessionStorage.clear();
```

Working with Cookies

- **Definition**: Cookies, also known as web cookies or browser cookies, are small pieces of data that a server sends to a user's web browser. These cookies are stored on the user's device and sent back to the server with subsequent requests. Cookies are essential in helping web applications maintain state and remember user information, which is especially important since HTTP is a stateless protocol.
- **Session Cookies**: These cookies only last for the duration of the user's session on the website. Once the user closes the browser or tab, the session cookie is deleted. These cookies are typically used for tasks like keeping a user logged in during their visit.
- **Secure Cookies**: These cookies are only sent over HTTPS, ensuring that they cannot be intercepted by an attacker in transit.
- **HttpOnly Cookies**: These cookies cannot be accessed or modified by JavaScript running in the browser, making them more secure against cross-site scripting (XSS) attacks.
- **Set-Cookie Header**: When you visit a website, the server can send a Set-Cookie header in the HTTP response. This header tells your browser to save a cookie with specific information. For example, it might store a unique ID that helps the site recognize you the next time you visit. You can manually set a cookie in JavaScript using `document.cookie`:

```
document.cookie = "organization=freeCodeCamp; expires=Fri, 31 Dec 2021
23:59:59 GMT; path=/";
```

To delete a cookie, you can set its expiration date to a time in the past.

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/";
```

Cache API

- **Definition**: Caching is the process of storing copies of files in a temporary storage location, so that they can be accessed more quickly. The Cache API is used to store network requests and responses, making web applications work more efficiently and even function offline. It is part of the broader Service Worker API and is crucial for creating Progressive Web Apps (PWAs) that can work under unreliable or slow network conditions. The Cache API is a storage mechanism that stores Request and Response objects. When a request is made to a server, the application can store the response

and later retrieve it from the cache instead of making a new network request. This reduces load times, saves bandwidth, and improves the overall user experience.

- **Cache Storage**: This is used to store key-value pairs of HTTP requests and their corresponding responses. This enables efficient retrieval of previously requested resources, reducing the need to fetch them from the network on subsequent visits and improving performance.
- **Cache-Control**: Developers can define how long a cached resource should be kept, and if it should be revalidated or served directly from cache.
- **Offline Support**: By using the Cache API, you can create offline-first web applications. For example, a PWA can serve cached assets when the user is disconnected from the network.

Negative Patterns and Client Side Storage

- **Excessive Tracking**: This refers to the practice of collecting and storing an overabundance of user data in client-side storage (such as cookies, local storage, or session storage) without clear, informed consent or a legitimate need. This often involves tracking user behavior, preferences, and interactions across multiple sites or sessions, which can infringe on user privacy.
- **Browser Fingerprinting**: A technique used to track and identify individual users based on unique characteristics of their device and browser, rather than relying on cookies or other traditional tracking methods. Unlike cookies, which are stored locally on a user's device, fingerprinting involves collecting a range of information that can be used to create a distinctive "fingerprint" of a user's browser session.
- **Setting Passwords in LocalStorage**: This might seem like a more obvious negative pattern, but setting any sensitive data like passwords in local storage posses a security risk. Local Storage is not encrypted and can be accessed easily. So you should never store any type of sensitive data in there.

IndexedDB

- **Definition**: IndexedDB is used for storing structured data in the browser. This is built into modern web browsers, allowing web apps to store and fetch JavaScript objects efficiently.

Cache/Service Workers

- **Definition**: A Service Worker is a script that runs in the background which is separate from your web page. It can intercept network requests, access the cache, and make the web app work offline. This is a key component of Progressive Web Apps.

Basics of Working with Classes

- **Definition**: Classes in JavaScript are used to define blueprints for creating objects, and encapsulating data. Classes include a constructor which is a special method that gets called automatically when a new object is created from the class. It is used to initialize the properties of the object. The `this` keyword is used here to refer to the current instance of the class. Below the constructor, you can have what are called methods. Methods are functions defined inside a class that perform actions or operations on the class's data or state. They are used to define behaviors that instances of the class can perform.

```
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log(`${this.name} says woof!`);
  }
}
```

To create a new instance of the class, you will use the `new` keyword followed by the class name:

```
const dog = new Dog("Gino");
```

You can also create classes as class expressions. This is where the class is anonymous and assigned to a variable.

```
const Dog = class {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log(`${this.name} says woof!`);
  }
};
```

Class Inheritance

- **Definition**: In programming, inheritance allows you to define classes that inherit properties and methods from parent classes. This promotes code reuse and establishes a hierarchical relationship between classes. A parent class is a class that acts like a blueprint for other classes. It defines properties and methods that are inherited by other classes. A child class is a class that inherits the properties and methods of another class. Child classes can also extend the functionality of their parent classes by adding new properties and methods. In JavaScript, we use the `extends` keyword to implement inheritance. This keyword indicates that a class is the child class of another class.

```
class Vehicle {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}

class Car extends Vehicle {
  honk() {
    console.log("Honk! Honk!");
  }
}
```

The `super` keyword is used to access the parent class's methods, constructors, and fields.

```
class Vehicle {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}

class Car extends Vehicle {
  constructor(brand, year, numDoors) {
    super(brand, year);
    this.numDoors = numDoors;
  }
}
```

Working with Static Methods and Static Properties

- **Static methods**: These methods are often used for utility functions that don't need access to the specific state of an object. They are defined within classes to encapsulate related functionality. Static methods are also helpful for implementing "factory" methods. A factory method is a method that you define in addition to the constructor to create objects based on specific criteria.

```
class Movie {
  constructor(title, rating) {
    this.title = title;
    this.rating = rating;
  }

  static compareMovies(movieA, movieB) {
    if (movieA.rating > movieB.rating) {
      console.log(`${movieA.title} has a higher rating.`);
    } else if (movieA.rating < movieB.rating) {
      console.log(`${movieB.title} has a higher rating.`);
    } else {
      console.log("These movies have the same rating.");
    }
  }
}

let movieA = new Movie("Movie A", 80);
let movieB = new Movie("Movie B", 45);

Movie.compareMovies(movieA, movieB);
```

- **Static Properties**: These properties are used to define values or attributes that are associated with a class itself, rather than with instances of the class. Static properties are shared across all instances of the class and can be accessed without creating an instance of the class.

```
class Car {
  // Static property
  static numberOfWheels = 4;

  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  // Instance method
  getCarInfo() {
    return `${this.make} ${this.model}`;
  }

  // Static method
  static getNumberOfWheels() {
    return Car.numberOfWheels;
  }
}

// Accessing static property directly from the class
console.log(Car.numberOfWheels);
```

- Recursion is programming concept that allows you to call a function repeatedly until a base-case is reached.

Here is an example of a recursive function that calculates the factorial of a number:

```
function findFactorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * findFactorial(n - 1);
}
```

In the above example, the `findFactorial` function is called recursively until `n` reaches `0`. When `n` is `0`, the base case is reached and the function returns `1`. The function then returns the product of `n` and the result of the recursive call to `findFactorial(n - 1)`.

- Recursion allows you to handle something with an unknown depth, such as deeply nested objects/arrays, or a file tree.
- A call stack is used to keep track of the function calls in a recursive function. Each time a function is called, it is added to the call stack. When the base case is reached, the function calls are removed from the stack.
- You should carefully define the base case as calling it indefinitely can cause your code to crash. This is because the recursion keeps piling more and more function calls till the system runs out of memory.
- Recursions find their uses in solving mathematical problems like factorial and Fibonacci, traversing trees and graphs, generating permutations and combinations and much more.

Pure vs Impure Functions

- A pure function is one that always produces the same output for the same input and doesn't have any side effects. Its output depends only on its input, and it doesn't modify any external state.
- Impure functions have side effects, which are changes to the state of the program that are observable outside the function.

Functional programming

- Functional Programming is an approach to software development that emphasizes the use of functions to solve problems, focusing on what needs to be done rather than how to do it.
- Functional programming encourages the use of techniques that help avoid side effects, such as using immutable data structures and higher-order functions.
- When used correctly, functional programming principles lead to cleaner and more maintainable code

Currying

- Currying is a functional programming technique that transforms a function with multiple arguments into a sequence of functions, each taking a single argument.

Here is an example of a regular function vs a curried function:

```
// Regular function

function average(a, b, c) {
  return (a + b + c) / 3;
}

// Curried function

function curriedAverage(a) {
  return function(b) {
    return function(c) {
      return (a + b + c) / 3;
    };
  };
```

```
}

// Usage of curried function

const avg = curriedAverage(2)(3)(4);
```

- Currying can be particularly powerful when working with functions that take many arguments.
- Currying makes your code more flexible and easier to reuse.
- You can use arrow functions to create curried functions more concisely:

```
const curriedAverage = a => b => c => (a + b + c) / 3;
```

- While currying can lead to more flexible and reusable code, it can also make code harder to read if overused.
- **Synchronous JavaScript** is executed sequentially and waits for the previous operation to finish before moving on to the next one.
- **Asynchronous JavaScript** allows multiple operations to be executed in the background without blocking the main thread.
- **Thread** is a sequence of instructions that can be executed independently of the main program flow.
- **Callback functions** are functions that are passed as arguments to other functions and are executed after the completion of the operation or as a result of an event.

The JavaScript engine and JavaScript runtime

- The **JavaScript engine** is a program that executes JavaScript code in a web browser. It works like a converter that takes your code, turns it into instructions that the computer can understand and work accordingly.
- V8 is an example of a JavaScript engine developed by Google.
- The **JavaScript runtime** is the environment in which JavaScript code is executed. It includes the JavaScript engine which processes and executes the code, and additional features like a web browser or Node.js.

The Fetch API

- The Fetch API allows web apps to make network requests, typically to retrieve or send data to the server. It provides a `fetch()` method that you can use to make these requests.
- You can retrieve text, images, audio, JSON, and other types of data using the Fetch API.

HTTP methods for Fetch API

The Fetch API supports various HTTP methods to interact with the server. The most common methods are:

- **GET**: Used to retrieve data from the server. By default, the Fetch API uses the `GET` method to retrieve data.

```
fetch('https://api.example.com/data')
```

To use the fetched data, it must be converted to JSON format using the `.json()` method:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
```

In this code, the response coming from the Fetch API is a promise and the `.then` handler is converting the response to a JSON format.

- **POST**: Used to send data to the server. The `POST` method is used to create new resources on the server.

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
```

In this example, we're sending a `POST` request to create a new user. We have specified the method as `POST`, set the appropriate headers, and included a body with the data we want to send. The body needs to be a string, so we use `JSON.stringify()` to convert our object to a JSON string.

- **PUT**: Used to update data on the server. The `PUT` method is used to update existing resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Smith',
    email: 'john@example.com'
  })
})
```

In this example, we are updating the ID `45` that is specified at the end of the URL. We have used the `PUT` method on the code and also specified the data as the body which will be used to update the identified data.

- **DELETE**: Used to delete data on the server. The `DELETE` method is used to delete resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'DELETE'
})
```

In this example, we're sending a `DELETE` request to remove a user with the ID `45`.

Promise and promise chaining

- **Promises** are objects that represent the eventual completion or failure of an asynchronous operation and its resulting value. The value of the promise is known only when the `async` operation is completed.
- Here is an example to create a simple promise:

```
```

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data received successfully');
  }, 2000);
});
```

- The `.then()` method is used in a Promise to specify what should happen when the Promise is fulfilled, while `.catch()` is used to handle any errors that occur.
- Here is an example of using `.then()` and `.catch()` with a Promise:

```
promise
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, the `.then()` method is used to log the data received from the Promise, while the `.catch()` method is used to log any errors that occur.

- **Promise chaining**: One of the powerful features of Promises is that we can chain multiple asynchronous operations together. Each `.then()` can return a new Promise, allowing you to perform a sequence of asynchronous operations one after the other.
- Here is an example of Promise chaining:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return fetch('https://api.example.com/other-data');
  })
  .then(response => response.json())
  .then(otherData => {
    console.log(otherData);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, we first fetch data from one URL, then fetch data from another URL based on the first response, and finally log the second data received.

The `catch` method would handle any errors that occur during the process. This means you don't need to add error handling to each individual step, which can greatly simplify your code.

Using `async/await` to handle promises

Async/await makes writing & reading asynchronous code easier which is built on top of Promises.

- **async**: The `async` keyword is used to define an asynchronous function. An `async` function returns a Promise, which resolves with the value returned by the `async` function.
- **await**: The `await` keyword is used inside an `async` function to pause the execution of the function until the Promise is resolved. It can only be used inside an `async` function.
- Here is an example of using `async/await`:

```
```

```
async function delayedGreeting(name) {
  console.log("A Messenger entered the chat...");
  await new Promise(resolve => setTimeout(resolve, 2000));
  console.log(`Hello, ${name}!`);
}

delayedGreeting("Alice");
console.log("First Printed Message!");
```

In the above example, the `delayedGreeting` function is an `async` function that pauses for 2 seconds before printing the greeting message. The `await` keyword is used to pause the function execution until the `Promise` is resolved.

- One of the biggest advantages of `async/await` is error handling via `try/catch` blocks. Here's an example:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

fetchData();
```

In the above example, the `try` block contains the code that might throw an error, and the `catch` block handles the error if it occurs. This makes error handling more straightforward and readable.

The `async` attribute

- The `async` attribute tells the browser to download the script file asynchronously while continuing to parse the HTML document.
- Once the script is downloaded, the HTML parsing is paused, the script is executed, and then HTML parsing resumes.
- You should use `async` for independent scripts where the order of execution doesn't matter

The `defer` attribute

- The `defer` attribute also downloads the script asynchronously, but it defers the execution of the script until after the HTML document has been fully parsed.
- The `defer` scripts maintain the order of execution as they appear in the HTML document.
- It's important to note that both `async` and `defer` attributes are ignored for inline scripts and only work for external script files.
- When both `async` and `defer` attributes are present, the `async` attribute takes precedence.

Geolocation API

- The Geolocation API provides a way for websites to request the user's location.
- The example below demonstrates the API's `getCurrentPosition()` method which is used to get the user's current location.

```
```

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log("Latitude: " + position.coords.latitude);
    console.log("Longitude: " + position.coords.longitude);
  },
  (error) => {
    console.log("Error: " + error.message);
  }
);
```

In this code, we're calling `getCurrentPosition` and passing it a function which will be called when the position is successfully obtained.

The `position` object contains a variety of information, but here we have selected `latitude` and `longitude` only.

If there is an issue with getting the `position`, then the error will be logged to the console. It is important to respect the user's privacy and only request their location when necessary.

--assignment--

Review the JavaScript topics and concepts.

JavaScript Arrays Review

JavaScript Array Basics

- **Definition**: A JavaScript array is an ordered collection of values, each identified by a numeric index. The values in a JavaScript array can be of different data types, including numbers, strings, booleans, objects, and even other arrays. Arrays are contiguous in memory, which means that all elements are stored in a single, continuous block of memory locations, allowing for efficient indexing and fast access to elements by their index.

```
const developers = ["Jessica", "Naomi", "Tom"];
```

- **Accessing Elements From Arrays**: To access elements from an array, you will need to reference the array followed by its index number inside square brackets. JavaScript arrays are zero based indexed which means the first element is at index 0, the second element is at index 1, etc. If you try to access an index that doesn't exist for the array, then JavaScript will return `undefined`.

```
const developers = ["Jessica", "Naomi", "Tom"];
developers[0] // "Jessica"
developers[1] // "Naomi"

developers[10] // undefined
```

- **`length` Property**: This property is used to return the number of items in an array.

```
const developers = ["Jessica", "Naomi", "Tom"];
developers.length // 3
```

- **Updating Elements in an Array**: To update an element in an array, you use the assignment operator (=) to assign a new value to the element at a specific index.

```
const fruits = ['apple', 'banana', 'cherry'];
fruits[1] = 'blueberry';

console.log(fruits); // ['apple', 'blueberry', 'cherry']
```

Two Dimensional Arrays

- **Definition**: A two-dimensional array is essentially an array of arrays. It's used to represent data that has a natural grid-like structure, such as a chessboard, a spreadsheet, or pixels in an image. To access an element in a two-dimensional array, you need two indices: one for the row and one for the column.

```
const chessboard = [
    ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
    ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
    ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']
];

console.log(chessboard[0][3]); // "Q"
```

Array Destructuring

- **Definition**: Array destructuring is a feature in JavaScript that allows you to extract values from arrays and assign them to variables in a more concise and readable way. It provides a convenient syntax for unpacking array elements into distinct variables.

```
const fruits = ["apple", "banana", "orange"];

const [first, second, third] = fruits;

console.log(first); // "apple"
console.log(second); // "banana"
console.log(third); // "orange"
```

- **Rest Syntax**: This allows you to capture the remaining elements of an array that haven't been destructured into a new array.

```
const fruits = ["apple", "banana", "orange", "mango", "kiwi"];
const [first, second, ...rest] = fruits;

console.log(first); // "apple"
console.log(second); // "banana"
console.log(rest); // ["orange", "mango", "kiwi"]
```

Common Array Methods

- `push()` **Method**: This method is used to add elements to the end of the array and will return the new length.

```
const desserts = ["cake", "cookies", "pie"];
desserts.push("ice cream");

console.log(desserts); // ["cake", "cookies", "pie", "ice cream"];
```

- `pop()` **Method**: This method is used to remove the last element from an array and will return that removed element. If the array is empty, then the return value will be `undefined`.

```
const desserts = ["cake", "cookies", "pie"];
desserts.pop();

console.log(desserts); // ["cake", "cookies"];
```

- `shift()` **Method**: This method is used to remove the first element from an array and return that removed element. If the array is empty, then the return value will be `undefined`.

```
const desserts = ["cake", "cookies", "pie"];
desserts.shift();

console.log(desserts); // ["cookies", "pie"];
```

- `unshift()` **Method**: This method is used to add elements to the beginning of the array and will return the new length.

```
const desserts = ["cake", "cookies", "pie"];
desserts.unshift("ice cream");

console.log(desserts); // ["ice cream", "cake", "cookies", "pie"];
```

- `indexOf()` **Method**: This method is useful for finding the first index of a specific element within an array. If the element cannot be found, then it will return `-1`.

```
const fruits = ["apple", "banana", "orange", "banana"];
const index = fruits.indexOf("banana");

console.log(index); // 1
console.log(fruits.indexOf("not found")); // -1
```

- `splice()` **Method**: This method is used to add or remove elements from any position in an array. The return value for the `splice()` method will be an array of the items removed from the array. If nothing is removed, then an empty array will be returned. This method will mutate the original array, modifying it in place rather than creating a new array. The first argument specifies the index at which to begin modifying the array. The second argument is the number of elements you wish to remove. The following arguments are the elements you wish to add.

```
const colors = ["red", "green", "blue"];
colors.splice(1, 0, "yellow", "purple");

console.log(colors); // ["red", "yellow", "purple", "green", "blue"]
```

- `includes()` **Method**: This method is used to check if an array contains a specific value. This method returns `true` if the array contains the specified element, and `false` otherwise.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];

console.log(programmingLanguages.includes("Python")); // true
console.log(programmingLanguages.includes("Perl")); // false
```

- `concat()` **Method**: This method creates a new array by merging two or more arrays.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];
const newList = programmingLanguages.concat("Perl");

console.log(newList); // ["JavaScript", "Python", "C++", "Perl"]
```

- `slice()` **Method**: This method returns a new array containing a shallow copy of a portion of the original array, specified by start and end indices. The new array contains references to the same elements as the original array (not duplicates). This means that if the elements are primitives (like numbers or strings), the values are copied; but if the elements are objects or arrays, the references are copied, not the objects themselves.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];
const newList = programmingLanguages.slice(1);

console.log(newList); // ["Python", "C++"]
```

- **Spread Syntax**: The spread syntax is used to create shallow copies of an array.

```
const originalArray = [1, 2, 3];
const shallowCopiedArray = [...originalArray];

shallowCopiedArray.push(4);

console.log(originalArray); // [1, 2, 3]
console.log(shallowCopiedArray); // [1, 2, 3, 4]
```

- `split()` **Method**: This method divides a string into an array of substrings and specifies where each split should happen based on a given separator. If no separator is provided, the method returns an array containing the original string as a single element.

```
const str = "hello";
const charArray = str.split("");

console.log(charArray); // ["h", "e", "l", "l", "o"]
```

- `reverse()` **Method**: This method reverses an array in place.

```
const desserts = ["cake", "cookies", "pie"];
console.log(desserts.reverse()); // ["pie", "cookies", "cake"]
```

- **`join()` Method**: This method concatenates all the elements of an array into a single string, with each element separated by a specified separator. If no separator is provided, or an empty string (`""`) is used, the elements will be joined without any separator.

```
const reversedArray = ["o", "l", "l", "e", "h"];
const reversedString = reversedArray.join("");

console.log(reversedString); // "olleh"
```

--assignment--

Review the JavaScript Arrays topics and concepts.

JavaScript Audio and Video Review

`Audio` Constructor and Common Methods

- **Definition**: The `Audio` constructor, like other constructors, is a special function called with the `new` keyword. It returns an `HTMLAudioElement`, which you can then use to play audio for the user or append to the DOM for the user to control themselves. When you call the constructor, you can optionally pass a URL as the (only) argument. This URL should point to the source of the audio file you want to play. Or, if you need to change the source dynamically, you can assign the URL to the `src` property of the returned audio element.
- **`play()` Method**: This method is used with the `audio` or `video` elements to begin playback for the media.

```
const audio = document.getElementById('audio');

// Starts playing the audio
audio.play();
```

- **`pause()` Method**: This method is used with the `audio` or `video` elements to pause playback for the media. It maintains the current position, so if `play()` is called, it starts from that position.

```
function pauseAudio() {
  const audio = document.getElementById('myAudio');
  audio.pause();  // Pauses the audio playback
}
```

- **`addTextTrack()` Method**: This method allows you to specify a text track to associate with the media element - which is especially helpful for adding subtitles to a video.
- **`fastSeek()` Method**: This method allows you to move the playback position to a specific time within the media.

The `Audio()` constructor has properties just as it has methods. Those properties include:

- `currentTime`: for getting the current playback time of an audio
- `loop`: for making the audio play continuously by automatically restarting when it reaches the end
- `muted`: for silencing the audio output of a media element regardless of volume setting

Different Audio and Video Formats

- **MIME type**: A MIME type, standing for Multipurpose Internet Mail Extensions, is a standardized way to programmatically indicate a file type. The MIME type can tell an application, such as your browser, how to handle a specific file. In the case of audio and video, the MIME type indicates it is a multimedia format that can be embedded in the web page.
- `source` **Element**: This is used to specify a file type and source - and can include multiple different types by using multiple source elements. When you do this, the browser will determine the best format to use for the user's current environment.
- **MP3**: This is a type of digital file format used to store music, audio, or sound. It's a compressed version of a sound recording that makes the file size smaller, so it's easier to store and share. MP3 has the widest browser support and the MIME type of `audio/mp3`.
- **MP4**: An MP4 is a type of digital file format used to store video and audio. It serves as a container that holds both the video (images) and the sound (music or speech) in one file. An MP4, can have the MIME type `audio/mp4` OR `video/mp4`, depending on whether it's a video file or audio-only.
- Other formats are `WMV` which is associated with the Windows Media Player app, `OGG`, `MKV`, `AVI`, and more.

codecs

- **Definition**: A codec, short for "encoder/decoder", is an algorithm or software that can convert audio and video between analogue and digital formats. Codecs can be specified as part of the MIME type. The basic syntax to define a codec is to add a semi-colon after the media type, then `codecs=` and the codec.

HTMLMediaElement API

- **Definition**: The `HTMLMediaElement` API is used to control the behavior of audio and video elements on your page. It extends the base `HTMLElement` interface, so you have access to the base properties, methods as well as these helpful events like `waiting`, `ended`, `canplay`, `canplaythrough`, and more. Examples of the methods include `play()`, `fastSeek()`, `pause()`, and `canPlayType()` for checking if a browser is likely to be able to play an audio file.

Media Capture and Streams API

- **Definition**: The Media Capture and Streams API, or the MediaStream API, is used to capture audio and video from your device. In order to use the API, you need to create the `MediaStream` object. You could do this with the constructor, but it would not be tied to the user's hardware. Instead, the `mediaDevices` property of the `global` navigator object has a `getUserMedia()` method for you to use.

```
window.navigator.mediaDevices.getUserMedia({
  audio: true,
  video: {
    width: {
      min: 1280,
      ideal: 1920,
      max: 3840
    },
    height: {
      min: 720,
      ideal: 1080,
      max: 2160
    }
  }
});
```

Screen Capture API

- **Definition**: The Screen Capture API is used to record a user's screen. This API is exposed by calling the `getDisplayMedia()` method of the `mediaDevices` object and consuming the returned media stream.

MediaStream Recording API

- **Definition**: The MediaStream Recording API works in tandem with the MediaStreams APIs, allowing you to record a MediaStream (or even an `HTMLMediaElement` directly).

Media Source Extensions API

- **topic**: The Media Source Extensions API is what allows you to directly pass a user's webcam feed to a video element with the `srcObject` property.

Web Audio API

- **Definition**: The Web Audio API which powers everything audible on the web. This API includes important objects like an `AudioBuffer` (representing a Buffer specifically containing audio data) or the `AudioContext`, used to represent an audio-processing graph.

--assignment--

Review the JavaScript Audio and Video topics and concepts.

JavaScript Classes Review

Basics of Working with Classes

- **Definition**: Classes in JavaScript are used to define blueprints for creating objects, and encapsulating data. Classes include a constructor which is a special method that gets called automatically when a new object is created from the class. It is used to initialize the properties of the object. The `this` keyword is used here to refer to the current instance of the class. Below the constructor, you can have what are called methods. Methods are functions defined inside a class that perform actions or operations on the class's data or state. They are used to define behaviors that instances of the class can perform.

```
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log(`${this.name} says woof!`);
  }
}
```

To create a new instance of the class, you will use the `new` keyword followed by the class name:

```
const dog = new Dog("Gino");
```

You can also create classes as class expressions. This is where the class is anonymous and assigned to a variable.

```
```

```
const Dog = class {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log(`${this.name} says woof!`);
  }
};
```

Class Inheritance

- **Definition**: In programming, inheritance allows you to define classes that inherit properties and methods from parent classes. This promotes code reuse and establishes a hierarchical relationship between classes. A parent class is a class that acts like a blueprint for other classes. It defines properties and methods that are inherited by other classes. A child class is a class that inherits the properties and methods of another class. Child classes can also extend the functionality of their parent classes by adding new properties and methods. In JavaScript, we use the `extends` keyword to implement inheritance. This keyword indicates that a class is the child class of another class.

```
class Vehicle {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}

class Car extends Vehicle {
  honk() {
    console.log("Honk! Honk!");
  }
}
```

The `super` keyword is used to access the parent class's methods, constructors, and fields.

```
class Vehicle {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}

class Car extends Vehicle {
  constructor(brand, year, numDoors) {
    super(brand, year);
    this.numDoors = numDoors;
  }
}
```

Working with Static Methods and Static Properties

- **Static methods**: These methods are often used for utility functions that don't need access to the specific state of an object. They are defined within classes to encapsulate related functionality. Static methods are also helpful for implementing "factory" methods. A factory method is a method that you define in addition to the constructor to create objects based on specific criteria.

```
```

```
class Movie {
  constructor(title, rating) {
    this.title = title;
    this.rating = rating;
  }

  static compareMovies(movieA, movieB) {
    if (movieA.rating > movieB.rating) {
      console.log(`${movieA.title} has a higher rating.`);
    } else if (movieA.rating < movieB.rating) {
      console.log(`${movieB.title} has a higher rating.`);
    } else {
      console.log("These movies have the same rating.");
    }
  }
}

let movieA = new Movie("Movie A", 80);
let movieB = new Movie("Movie B", 45);

Movie.compareMovies(movieA, movieB);
```

- **Static Properties**: These properties are used to define values or attributes that are associated with a class itself, rather than with instances of the class. Static properties are shared across all instances of the class and can be accessed without creating an instance of the class.

```
class Car {
  // Static property
  static numberOfWheels = 4;

  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  // Instance method
  getCarInfo() {
    return `${this.make} ${this.model}`;
  }

  // Static method
  static getNumberOfWheels() {
    return Car.numberOfWheels;
  }
}

// Accessing static property directly from the class
console.log(Car.numberOfWheels);
```

--assignment--

Review the JavaScript Classes topics and concepts.

JavaScript Comparisons and Conditionals Review

Comparisons and the `null` and `undefined` Data Types

- **Comparisons and `undefined`**: A variable is `undefined` when it has been declared but hasn't been assigned a value. It's the default value of uninitialized variables and function parameters that weren't provided an argument. `undefined` converts to `NaN` in numeric contexts, which makes all numeric comparisons with `undefined` return `false`.

```
console.log(undefined > 0);  // false
console.log(undefined < 0);  // false
console.log(undefined == 0); // false
```

- **Comparisons and `null`**: The `null` type represents the intentional absence of a value. When using the equality operator, `null` and `undefined` are considered equal. However, when using the strict equality operator (`===`), which checks both value and type without performing type coercion, `null` and `undefined` are not equal:

```
console.log(null == undefined); // true
console.log(null === undefined); // false
```

`switch` Statements

- **Definition**: A `switch` statement evaluates an expression and matches its value against a series of `case` clauses. When a match is found, the code block associated with that case is executed. A `break` statement should be placed at the end of each case, to terminate its execution and continue with the next. The `default` case is an optional case and only executes if none of the other cases match. The `default` case is placed at the end of a `switch` statement.

```
const dayOfWeek = 3;

switch (dayOfWeek) {
  case 1:
    console.log("It's Monday! Time to start the week strong.");
    break;
  case 2:
    console.log("It's Tuesday! Keep the momentum going.");
    break;
  case 3:
    console.log("It's Wednesday! We're halfway there.");
    break;
  case 4:
    console.log("It's Thursday! Almost the weekend.");
    break;
  case 5:
    console.log("It's Friday! The weekend is near.");
    break;
  case 6:
    console.log("It's Saturday! Enjoy your weekend.");
    break;
  case 7:
    console.log("It's Sunday! Rest and recharge.");
    break;
  default:
    console.log("Invalid day! Please enter a number between 1 and 7.");
}
```

--assignment--

Review the JavaScript Comparisons and Conditionals topics and concepts.

JavaScript Dates Review

The `date()` Object and Common Methods

- **Definition**: The `date()` object is used to create, manipulate, and format dates and times in JavaScript. In the following example, the `new` keyword is used to create a new instance of the `Date` object, and the `Date` object is then assigned to the variable `now`. If you were to log the value of `now` to the console, you would see the current date and time based on the system clock of the computer running the code.

```
const now = new Date();
```

- **`Date.now()` Method**: This method is used to get the current date and time. `Date.now()` returns the number of milliseconds since January 1, 1970, 00:00:00 UTC. This is known as the Unix epoch time. Unix epoch time is a common way to represent dates and times in computer systems because it is an integer that can be easily stored and manipulated. UTC stands for Universal Time Coordinated, which is the primary time standard by which the world regulates clocks and time.
- **`getDate()` Method**: This method is used to get a day of the month based on the current date. `getDate()` will return an integer value between 1 and 31, depending on the day of the month. If the date is invalid, it will return `NaN` (Not a Number).

```
const now = new Date();
const date = now.getDate();
console.log(date); // 15
```

- **`getMonth()` Method**: This method is used to get the month. The month is zero-based, so January is 0, February is 1, and so on. In this example, the output is 2, which corresponds to March. If the month is invalid, it will return `NaN`.

```
const now = new Date();
const month = now.getMonth();
console.log(month); // 2
```

- **`getFullYear()` Method**: This method is used to get the full year. If the year is invalid, it will return `NaN`.

```
const now = new Date();
const year = now.getFullYear();
console.log(year); // 2024
```

Different Ways to Format Dates

- **`toISOString()` Method**: This method is used to format the date in an extended `ISO` (ISO 8601) format. ISO 8601 is an international standard for representing dates and times. The format is `YYYY-MM-DDTHH:mm:ss.sssZ`.

```
const date = new Date();
console.log(date.toISOString());
```

- `toLocaleDateString()` **Method**: This method is used to format the date based on the user's locale.

```
const date = new Date();
console.log(date.toLocaleDateString());  // 11/23/2024
```

The `toLocaleDateString()` method accepts two optional parameters: locales and options.

The locales parameter is a string representing the locale to use. For example, you can pass in `"en-US"` for English (United States) or `"fr-FR"` for French (France). If you don't pass in a locales parameter, the default locale is used. The second optional parameter is the options parameter. This parameter is an object that allows you to specify the format of the date string.

```
const date = new Date();
const options = {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric",
};
console.log(date.toLocaleDateString("en-GB", options)); // Saturday, November
23, 2024
```

--assignment--

Review the JavaScript Dates topics and concepts.

JavaScript Functional Programming Review

Pure vs Impure Functions

- A pure function is one that always produces the same output for the same input and doesn't have any side effects. Its output depends only on its input, and it doesn't modify any external state.
- Impure functions have side effects, which are changes to the state of the program that are observable outside the function.

Functional programming

- Functional Programming is an approach to software development that emphasizes the use of functions to solve problems, focusing on what needs to be done rather than how to do it.
- Functional programming encourages the use of techniques that help avoid side effects, such as using immutable data structures and higher-order functions.
- When used correctly, functional programming principles lead to cleaner and more maintainable code

Currying

- Currying is a functional programming technique that transforms a function with multiple arguments into a sequence of functions, each taking a single argument.

Here is an example of a regular function vs a curried function:

```
// Regular function

function average(a, b, c) {
  return (a + b + c) / 3;
}

// Curried function

function curriedAverage(a) {
  return function(b) {
    return function(c) {
      return (a + b + c) / 3;
    };
  };
}

// Usage of curried function

const avg = curriedAverage(2)(3)(4);
```

- Currying can be particularly powerful when working with functions that take many arguments.
- Currying makes your code more flexible and easier to reuse.
- You can use arrow functions to create curried functions more concisely:

```
const curriedAverage = a => b => c => (a + b + c) / 3;
```

- While currying can lead to more flexible and reusable code, it can also make code harder to read if overused.

--assignment--

Review the JavaScript Functional Programming topics and concepts.

JavaScript Functions Review

JavaScript Functions

- Functions are reusable blocks of code that perform a specific task.
- Functions can be defined using the `function` keyword followed by a name, a list of parameters, and a block of code that performs the task.

```
function addNumbers(x, y, z) {
  return x + y + z;
}

console.log(addNumbers(5, 3, 8)); // Output: 16
```

- Arguments are values passed to a function when it is called.
- A function call is the process of executing a function in a program by specifying the function's name followed by parentheses, optionally including arguments inside the parentheses.
- When a function finishes its execution, it will always return a value.
- By default, the return value of a function is `undefined`.

- The `return` keyword is used to specify the value to be returned from the function and ends the function execution.
- Default parameters allow functions to have predefined values that will be used if an argument is not provided when the function is called. This makes functions more flexible and prevents errors in cases where certain arguments might be omitted.

```
const calculateTotal = (amount, taxRate = 0.05) => {
  return amount + (amount * taxRate);
};

console.log(calculateTotal(100)); // Output: 105
```

- Function Expressions are functions that you assign to variables. By doing this, you can use the function in any part of your code where the variable is accessible.

```
const multiplyNumbers = function(firstNumber, secondNumber) {
  return firstNumber * secondNumber;
};

console.log(multiplyNumbers(4, 5)); // Output: 20
```

Arrow Functions

- Arrow functions are a more concise way to write functions in JavaScript.

```
const calculateArea = (length, width) => {
  const area = length * width;
  return `The area of the rectangle is ${area} square units.`;
};

console.log(calculateArea(5, 10)); // Output: "The area of the rectangle is
50 square units."
```

- When defining an arrow function, you do not need the `function` keyword.
- If you are using a single parameter, you can omit the parentheses around the parameter list.

```
const cube = x => {
  return x * x * x;
};

console.log(cube(3)); // Output: 27
```

- If the function body consists of a single expression, you can omit the curly braces and the `return` keyword.

```
const square = number => number * number;

console.log(square(5)); // Output: 25
```

Scope in Programming

- **Global scope**: This is the outermost scope in JavaScript. Variables declared in the global scope are accessible from anywhere in the code and are called global variables.

- **Local scope**: This refers to variables declared within a function. These variables are only accessible within the function where they are declared and are called local variables.
- **Block scope**: A block is a set of statements enclosed in curly braces `{}` such as in `if` statements, or loops.
- Block scoping with `let` and `const` provides even finer control over variable accessibility, helping to prevent errors and make your code more predictable.

--assignment--

Review the JavaScript Functions topics and concepts.

JavaScript Fundamentals Review

String Constructor and `toString()` Method

- **Definition**: A string object is used to represent a sequence of characters. String objects are created using the `String` constructor function, which wraps the primitive value in an object.

```
const greetingObject = new String("Hello, world!");

console.log(typeof greetingObject); // "object"
```

- **`toString()` Method**: This method converts a value to its string representation. It is a method you can use for numbers, booleans, arrays, and objects.

```
const num = 10;
console.log(num.toString()); // "10"

const arr = [1, 2, 3];
console.log(arr.toString()); // "1,2,3"
```

This method accepts an optional radix which is a number from 2 to 36. This radix represents the base, such as base 2 for binary or base 8 for octal. If the radix is not specified, it defaults to base 10, which is decimal.

```
const num = 10;
console.log(num.toString(2)); // "1010"(binary)
```

Number Constructor

- **Definition**: The `Number` constructor is used to create a number object. The number object contains a few helpful properties and methods like the `isNaN` and `toFixed` method. Most of the time, you will be using the `Number` constructor to convert other data types to the number data type.

```
const myNum = new Number("34");
console.log(typeof myNum); // "object"

const num = Number('100');
console.log(num); // 100

console.log(typeof num); // number
```

Best Practices for Naming Variables and Functions

- **camelCasing**: By convention, JavaScript developers will use camel casing for naming variables and functions. Camel casing is where the first word is all lowercase and the following words start with a capital letter. Ex. `isLoading`.
- **Naming Booleans**: For boolean variables, it's a common practice to use prefixes such as "is", "has", or "can".

```
let isLoading = true;
let hasPermission = false;
let canEdit = true;
```

- **Naming Functions**: For functions, the name should clearly indicate what the function does. For functions that return a boolean (often called predicates), you can use the same "is", "has", or "can" prefixes. When you have functions that retrieve data, it is common to start with the word "get". When you have functions that set data, it is common to start with the word "set". For event handler functions, you might prefix with "handle" or suffix with "Handler".

```
function getUserData() { /* ... */ }

function isValidEmail(email) { /* ... */ }

function getProductDetails(productId) { /* ... */ }

function setUserPreferences(preferences) { /* ... */ }

function handleClick() { /* ... */ }
```

- **Naming Variables Inside Loops**: When naming iterator variables in loops, it's common to use single letters like `i`, `j`, or `k`.

```
for (let i = 0; i < array.length; i++) { /* ... */ }
```

Working with Sparse Arrays

- **Definition**: It is possible to have arrays with empty slots. Empty slots are defined as slots with nothing in them. This is different than array slots with the value of `undefined`. These types of arrays are known as sparse arrays.

```
const sparseArray = [1, , , 4];
console.log(sparseArray.length); // 4
```

Linters and Formatters

- **Linters**: A linter is a static code analysis tool that flags programming errors, bugs, stylistic errors, and suspicious constructs. An example of a common linter would be ESLint.
- **Formatters**: Formatters are tools that automatically format your code to adhere to a specific style guide. An example of a common formatter would be Prettier.

Memory Management

- **Definition**: Memory management is the process of controlling the memory, allocating it when needed and freeing it up when it's no longer needed. JavaScript uses automatic memory

management. This means that JavaScript (more specifically, the JavaScript engine in your web browser) takes care of memory allocation and deallocation for you. You don't have to explicitly free up memory in your code. This automatic process is often called "garbage collection."

Closures

- **Definition**: A closure is a function that has access to variables in its outer (enclosing) lexical scope, even after the outer function has returned.

```
function outerFunction(x) {
  let y = 10;
  function innerFunction() {
    console.log(x + y);
  }
  return innerFunction;
}

let closure = outerFunction(5);
closure(); // 15
```

`var` Keyword and Hoisting

- **Definition**: `var` was the original way to declare variables before 2015. But there were some issues that came with `var` in terms of scope, redeclaration and more. So that is why modern JavaScript programming uses `let` and `const` instead.
- **Redeclaring Variables with `var`**: If you try to redeclare a variable using `let`, then you would get a `SyntaxError`. But with `var`, you are allowed to redeclare a variable.

```
// Uncaught SyntaxError: Identifier 'num' has already been declared
let num = 19;
let num = 18;

var myNum = 5;
var myNum = 10; // This is allowed and doesn't throw an error

console.log(myNum) // 10
```

- **`var` and Scope**: Variables declared with `var` inside a block (like an `if` statement or a `for` loop) are still accessible outside that block.

```
if (true) {
  var num = 5;
}
console.log(num); // 5
```

- **Hoisting**: This is JavaScript's default behavior of moving declarations to the top of their respective scopes during the compilation phase before the code is executed. When you declare a variable using the `var` keyword, JavaScript hoists the declaration to the top of its scope.

```
console.log(num); // undefined
var num = 5;
console.log(num); // 5
```

When you declare a function using the function declaration syntax, both the function name and the function body are hoisted. This means you can call a function before you've declared it in your code.

```
sayHello(); // "Hello, World!"

function sayHello() {
  console.log("Hello, World!");
}
```

Variable declarations made with `let` or `const` are hoisted, but they are not initialized, and you can't access them before the actual declaration in your code. This behavior is often referred to as the "temporal dead zone".

```
console.log(num); // Throws a ReferenceError
let num = 10;
```

Working with Imports, Exports and Modules

- **Module**: This is a self-contained unit of code that encapsulates related functions, classes, or variables. To create a module, you write your JavaScript code in a separate file.
- **Exports**: Any variables, functions, or classes you want to make available to other parts of your application need to be explicitly exported using the `export` keyword. There are two types of export: named export and default export.
- **Imports**: To use the exported items in another part of your application, you need to import them using the `import` keyword. The types can be named import, default import, and namespace import.

```
// Within a file called math.js, we export the following functions:

// Named export
export function add(num1, num2) {
  return num1 + num2;
}

// Default export
export default function subtract(num1, num2) {
  return num1 - num2;
}

// Within another file, we can import the functions from math.js.

// Named import - This line imports the add function.
// The name of the function must exactly match the one exported from math.js.
import { add } from './math.js';

// Default import - This line imports the subtract function.
// The name of the function can be anything.
import subtractFunc from './math.js';

// Namespace import - This line imports everything from the file.
import * as Math from './math.js';

console.log(add(5, 3)); // 8
console.log(subtractFunc(5, 3)); // 2
console.log(Math.add(5, 3)); // 8
console.log(Math.subtract(5, 3)); // 2
```

--assignment--

Review the JavaScript Fundamentals topics and concepts.

JavaScript Higher Order Functions Review

Callback Functions and the `forEach` Method

- **Definition**: In JavaScript, a callback function is a function that is passed as an argument to another function and is executed after the main function has finished its execution.
- **`forEach()` Method**: This method is used to iterate over each element in an array and perform an operation on each element. The callback function in `forEach` can take up to three arguments: the current element, the index of the current element, and the array that `forEach` was called upon.

```
const numbers = [1, 2, 3, 4, 5];

// Result: 2 4 6 8 10
numbers.forEach((number) => {
  console.log(number * 2);
});
```

Higher Order Functions

- **Definition**: A higher order function takes one or more functions for the arguments and returns a function or value for the result.

```
function operateOnArray(arr, operation) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(operation(arr[i]));
  }
  return result;
}

function double(x) {
  return x * 2;
}

const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = operateOnArray(numbers, double);
console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

- **`map()` Method**: This method is used to create a new array by applying a given function to each element of the original array. The callback function can accept up to three arguments: the current element, the index of the current element, and the array that `map` was called upon.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((num) => num * 2);

console.log(numbers); // [1, 2, 3, 4, 5]
console.log(doubled); // [2, 4, 6, 8, 10]
```

- **`filter()` Method**: This method is used to create a new array with elements that pass a specified test, making it useful for selectively extracting items based on criteria. Just like the `map` method, the callback function for the `filter` method accepts the same three arguments: the current element being processed, the index, and the array.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const evenNumbers = numbers.filter((num) => num % 2 === 0);

console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

- **`reduce()` Method**: This method is used to process an array and condense it into a single value. This single value can be a number, a string, an object, or even another array. The `reduce()` method works by applying a function to each element in the array, in order, passing the result of each calculation on to the next. This function is often called the reducer function. The reducer function takes two main parameters: an accumulator and the current value. The accumulator is where you store the running result of your operations, and the current value is the array element being processed.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
  0
);

console.log(sum); // 15
```

Method Chaining

- **Definition**: Method chaining is a programming technique that allows you to call multiple methods on the same object in a single line of code. This technique can make your code more readable and concise, especially when performing a series of operations on the same object.

```
const result = "  Hello, World!  "
  .trim()
  .toLowerCase()
  .replace("world", "JavaScript");

console.log(result); // "hello, JavaScript!"
```

Working with the `sort` Method

- **Definition**: The `sort` method is used to sort the elements of an array and return a reference to the sorted array. No copy is made in this case because the elements are sorted in place.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();

console.log(fruits); // ["Apple", "Banana", "Mango", "Orange"]
```

If you need to sort numbers, then you will need to pass in a compare function. The `sort` method converts the elements to strings and then compares their sequences of UTF-16 code units values. UTF-16 code units are the numeric values that represent the characters in the string. Examples of UTF-16 code units are the numbers 65, 66, and 67 which represent the characters "A", "B", and "C" respectively. So the number 200

appears before the number 3 in an array, because the string "200" comes before the string "3" when comparing their UTF-16 code units.

```
const numbers = [414, 200, 5, 10, 3];

numbers.sort((a, b) => a - b);

console.log(numbers); // [3, 5, 10, 200, 414]
```

The parameters `a` and `b` are the two elements being compared. The compare function should return a negative value if `a` should come before `b`, a positive value if `a` should come after `b`, and zero if `a` and `b` are equal.

Working with the `every` and `some` Methods

- **`every()` Method**: This method tests whether all elements in an array pass a test implemented by a provided function. The `every()` method returns `true` if the provided function returns `true` for all elements in the array. If any element fails the test, the method immediately returns `false` and stops checking the remaining elements.

```
const numbers = [2, 4, 6, 8, 10];
const hasAllEvenNumbers = numbers.every((num) => num % 2 === 0);

console.log(hasAllEvenNumbers); // true
```

- **`some()` Method**: This method checks if at least one element passes the test. The `some()` method returns `true` as soon as it finds an element that passes the test. If no elements pass the test, it returns `false`.

```
const numbers = [1, 3, 5, 7, 8, 9];
const hasSomeEvenNumbers = numbers.some((num) => num % 2 === 0);

console.log(hasSomeEvenNumbers); // true
```

--assignment--

Review the JavaScript Higher Order Functions topics and concepts.

JavaScript Loops Review

Working with Loops

- **`for` Loop**: This type of loop is used to repeat a block of code a certain number of times. This loop is broken up into three parts: the initialization statement, the condition, and the increment/decrement statement. The initialization statement is executed before the loop starts. It is typically used to initialize a counter variable. The condition is evaluated before each iteration of the loop. An iteration is a single pass through the loop. If the condition is `true`, the code block inside the loop is executed. If the condition is `false`, the loop stops and you move on to the next block of code. The increment/decrement statement is executed after each iteration of the loop. It is typically used to increment or decrement the counter variable.

```
```

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

- **`for...of` Loop**: This type of loop is used when you need to loop over values from an iterable. Examples of iterables are arrays and strings.

```
const numbers = [1, 2, 3, 4, 5];

for (const num of numbers) {
  console.log(num);
}
```

- **`for...in` Loop**: This type of loop is best used when you need to loop over the properties of an object. This loop will iterate over all enumerable properties of an object, including inherited properties and non-numeric properties.

```
const fruit = {
  name: 'apple',
  color: 'red',
  price: 0.99
};

for (const prop in fruit) {
  console.log(fruit[prop]);
}
```

- **`while` Loop**: This type of loop will run a block of code as long as the condition is `true`.

```
let i = 5;

while (i > 0) {
  console.log(i);
  i--;
}
```

- **`do...while` Loop**: This type of loop will execute the block of code at least once before checking the condition.

```
let userInput;

do {
  userInput = prompt("Please enter a number between 1 and 10");
} while (Number(userInput) < 1 || Number(userInput) > 10);

alert("You entered a valid number!");
```

`break` and `continue` Statements

- **Definition**: A `break` statement is used to exit a loop early, while a `continue` statement is used to skip the current iteration of a loop and move to the next one.

```
```

```
// Example of break statement
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}

// Output: 0, 1, 2, 3, and 4

// Example of continue statement
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    continue;
  }
  console.log(i);
}

// Output: 0, 1, 2, 3, 4, 6, 7, 8, and 9
```

--assignment--

Review the JavaScript Loops topics and concepts.

JavaScript Maps and Sets Review

Sets in JavaScript

- A `Set` is a built-in option for managing data collection.
- Sets ensure that each value in it appears only once, making it useful for eliminating duplicates from an array or handling collections of distinct values.
- You can create a `Set` using the `Set()` constructor:

```
const set = new Set([1, 2, 3, 4, 5]);
console.log(set); // Set { 1, 2, 3, 4, 5 }
```

- Sets can be manipulated using these methods:
  - `add()`: Adds a new element to the `Set`.
  - `delete()`: Removes an element from the `Set`.
  - `has()`: Checks if an element exists in the `Set`.
  - `clear()`: Removes all elements from the `Set`.
  - `keys()` and `values()`: Both returns a `SetIterator` that contains the values of the `Set`. They are the same because `keys()` is an alias for `values()`.
  - `forEach()`: for iterating over the values of the `Set`.

Weaksets in JavaScript

- `WeakSet` is a collection of objects that allows you to store weakly held objects.

Sets vs WeakSets

- Unlike Sets, a `WeakSet` does not support primitives like numbers or strings.
- A `WeakSet` only stores objects, and the references to those objects are "weak," meaning that if the object is not being used anywhere else in your code, it is removed automatically to free up memory.

Maps in JavaScript

- A `Map` is a built-in object that holds key-value pairs just like an object.
- Maps differ from the standard JavaScript objects with their ability to allow keys of any type, including objects, and functions.
- A `Map` provides better performance over the standard object when it comes to frequent addition and removals of key-value pairs.
- You can create a `Map` using the `Map()` constructor:

```
const map = new Map([
  ['flower', 'rose'],
  ['fruit', 'apple'],
  ['vegetable', 'carrot']
]);
console.log(map); // Map(3) { 'flower' => 'rose', 'fruit' => 'apple',
'vegetable' => 'carrot' }
```

- Maps can be manipulated using these methods:
    - `set()`: Adds a new key-value pair to the `Map`.
    - `get()`: Retrieves the value of a key from the `Map`.
    - `delete()`: Removes a key-value pair from the `Map`.
    - `has()`: Checks if a key exists in the `Map`.
    - `clear()`: Removes all key-value pairs from the `Map`.

Note that both Maps and Sets have the `size` property that returns the number of unique elements in them.

WeakMaps in JavaScript

- A `WeakMap` is a collection of key-value pairs just like `Map`, but with weak references to the keys. The keys must be an object and the values can be anything you like.

Maps vs WeakMaps

- WeakMaps are similar to WeakSets in that they only store objects and the references to those objects are "weak".

--assignment--

Review the JavaScript Maps, Sets, and JSON topics and concepts.

JavaScript Math Review

Working with the Number Data Type

- **Definition**: JavaScript's `Number` type includes integers, floating-point numbers, `Infinity` and `NaN`. Floating-point numbers are numbers with a decimal point. Positive `Infinity` is a number greater than any other number while `-Infinity` is a number smaller than any other number. `NaN` (`Not a Number`) represents an invalid numeric value like the string `"Jessica"`.

Common Arithmetic Operations

- **Addition Operator**: This operator (`+`) is used to calculate the sum of two or more numbers.
- **Subtraction Operator**: This operator (`-`) is used to calculate the difference between two numbers.
- **Multiplication Operator**: This operator (`*`) is used to calculate the product of two or more numbers.

- **Division Operator**: This operator (`/`) is used to calculate the quotient between two numbers
- **Division By Zero**: If you try to divide by zero, JavaScript will return `Infinity`.
- **Remainder Operator**: This operator(`%`) returns the remainder of a division.
- **Exponentiation Operator**: This operator (`**`) raises one number to the power of another.

Calculations with Numbers and Strings

- **Explanation**: When you use the `+` operator with a number and a string, JavaScript will coerce the number into a string and concatenate the two values. When you use the `-`, `*` or `/` operators with a string and number, JavaScript will coerce the string into a number and the result will be a number. For `null` and `undefined`, JavaScript treats `null` as 0 and undefined as `NaN` in mathematical operations.

```
const result = 5 + '10';

console.log(result); // 510
console.log(typeof result); // string

const subtractionResult = '10' - 5;
console.log(subtractionResult); // 5
console.log(typeof subtractionResult); // number

const multiplicationResult = '10' * 2;
console.log(multiplicationResult); // 20
console.log(typeof multiplicationResult); // number

const divisionResult = '20' / 2;
console.log(divisionResult); // 10
console.log(typeof divisionResult); // number

const result1 = null + 5;
console.log(result1); // 5
console.log(typeof result1); // number

const result2 = undefined + 5;
console.log(result2); // NaN
console.log(typeof result2); // number
```

Operator Precedence

- **Definition**: Operator precedence determines the order in which operations are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. Values inside the parenthesis will be evaluated first and multiplication/division will have higher precedence than addition/subtraction. If the operators have the same precedence, then JavaScript will use associativity.

```
const result = (2 + 3) * 4;

console.log(result); // 20

const result2 = 10 - 2 + 3;

console.log(result2); // 11

const result3 = 2 ** 3 ** 2;

console.log(result3); // 512
```

- **Definition**: Associativity informs us the direction in which an expression is evaluated when multiple operators of the same type exist. It defines whether the expression should be evaluated from left-to-right (`left-associative`) or right-to-left (`right-associative`). For example, the exponent operator is also right to left associative:

```
const result4 = 5 ** 4 ** 1; // 625

console.log(result4);
```

Increment and Decrement Operators

- **Increment Operator**: This operator is used to increase the value by one. The prefix notation `++num` increases the value of the variable first, then returns a new value. The postfix notation `num++` returns the current value of the variable first, then increases it.

```
let x = 5;

console.log(++x); // 6
console.log(x); // 6


let y = 5;

console.log(y++); // 5
console.log(y); // 6
```

- **Decrement Operator**: This operator is used to decrease the value by one. The prefix notation and postfix notation work the same way as earlier with the increment operator.

```
let num = 5;

console.log(--num); // 4
console.log(num--); // 4
console.log(num); // 3
```

Compound Assignment Operators

- **Addition Assignment (`+=`) Operator**: This operator performs addition on the values and assigns the result to the variable.
- **Subtraction Assignment (`-=`) Operator**: This operator performs subtraction on the values and assigns the result to the variable.
- **Multiplication Assignment (`*=`) Operator**: This operator performs multiplication on the values and assigns the result to the variable.
- **Division Assignment (`/=`) Operator**: This operator performs division on the values and assigns the result to the variable.
- **Remainder Assignment (`%=`) Operator**: This operator divides a variable by the specified number and assigns the remainder to the variable.
- **Exponentiation Assignment (`**=`) Operator**: This operator raises a variable to the power of the specified number and reassigns the result to the variable.

Booleans and Equality

- **Boolean Definition**: A boolean is a data type that can only have two values: `true` or `false`.
- **Equality (`==`) Operator**: This operator uses type coercion before checking if the values are equal.

```
console.log(5 == '5'); // true
```

- **Strict Equality (`===`) Operator**: This operator does not perform type coercion and checks if both the types and values are equal.

```
console.log(5 === '5'); // false
```

- **Inequality (`!=`) Operator**: This operator uses type coercion before checking if the values are not equal.
- **Strict Inequality (`!==`) Operator**: This operator does not perform type coercion and checks if both the types and values are not equal.

Comparison Operators

- **Greater Than (`>`) Operator**: This operator checks if the value on the left is greater than the one on the right.
- **Greater Than (`>=`) or Equal Operator**: This operator checks if the value on the left is greater than or equal to the one on the right.
- **Less Than (`<`) Operator**: This operator checks if the value on the left is less than the one on the right.
- **Less Than (`<=`) or Equal Operator**: This operator checks if the value on the left is less than or equal to the one on the right.

Unary Operators

- **Unary Plus Operator**: This operator converts its operand into a number. If the operand is already a number, it remains unchanged.

```
const str = '42';
const num = +str;

console.log(num); // 42
console.log(typeof num); // number
```

- **Unary Negation (`-`) Operator**: This operator negates the operand.

```
const num = 4;
console.log(-num); // -4
```

- **Logical NOT (`!`) Operator**: This operator flips the boolean value of its operand. So, if the operand is `true`, it becomes `false`, and if it's `false`, it becomes `true`.

Bitwise Operators

- **Bitwise AND (`&`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of both operands are 1.
- **Bitwise AND Assignment (`&=`) Operator**: This operator performs a `bitwise AND` operation with the specified number and reassigns the result to the variable.
- **Bitwise OR (`|`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of either or both operands are 1.
- **Bitwise OR Assignment (`|=`) Operator**: This operator performs a `bitwise OR` operation with the specified number and reassigns the result to the variable.

- **Bitwise XOR (`^`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of either, but not both, operands are 1.
- **Bitwise NOT (`~`) Operator**: This operator inverts the binary representation of a number.
- **Left Shift (`<<`) Operator**: This operator shifts all bits to the left by a specified number of positions.
- **Right Shift (`>>`) Operator**: This operator shifts all bits to the right.

Conditional Statements, Truthy Values, Falsy Values and the Ternary Operator

- **`if/else if/else`**: An `if` statement takes a condition and runs a block of code if that condition is `truthy`. If the condition is `false`, then it moves to the `else if` block. If none of those conditions are `true`, then it will execute the `else` clause. `Truthy` values are any values that result in `true` when evaluated in a Boolean context like an `if` statement. `Falsy` values are values that evaluate to `false` in a Boolean context.

```
const score = 87;

if (score >= 90) {
 console.log('You got an A');
} else if (score >= 80) {
 console.log('You got a B'); // You got an B
} else if (score >= 70) {
 console.log('You got a C');
} else {
 console.log('You failed! You need to study more!');
}
```

- **Ternary Operator**: This operator is often used as a shorter way to write `if else` statements.

```
const temperature = 30;
const weather = temperature > 25 ? 'sunny' : 'cool';

console.log(`It's a ${weather} day!`); // It's a sunny day!
```

Binary Logical Operators

- **Logical AND (`&&`) Operator**: This operator checks if both operands are `true`. If both are `true`, then it will return the second value. If either operand is `falsy`, then it will return the `falsy` value. If both operands are `falsy`, it will return the first `falsy` value.

```
const result = true && 'hello';

console.log(result); // hello
```

- **Logical OR (`||`) Operator**: This operator checks if at least one of the operands is truthy.
- **Nullish Coalescing (`??`) Operator**: This operator will return a value only if the first one is `null` or `undefined`.

```
const userSettings = {
 theme: null,
 volume: 0,
 notifications: false,
};
```

```
let theme = userSettings.theme ?? 'light';
console.log(theme); // light
```

The `Math` Object

- **The `Math.random()` Method**: This method generates a random floating-point number between 0 (inclusive) and 1 (exclusive). This means the possible output can be 0, but it will never actually reach 1.
- **The `Math.max()` Method**: This method takes a set of numbers and returns the maximum value.
- **The `Math.min()` Method**: This method takes a set of numbers and returns the minimum value.
- **The `Math.ceil()` Method**: This method rounds a value up to the nearest whole integer.
- **The `Math.floor()` Method**: This method rounds a value down to the nearest whole integer.
- **The `Math.round()` Method**: This method rounds a value to the nearest whole integer.

```
console.log(Math.round(2.3)); // 2
console.log(Math.round(4.5)); // 5
console.log(Math.round(4.8)); // 5
```

- **The `Math.trunc()` Method**: This method removes the decimal part of a number, returning only the integer portion, without rounding.
- **The `Math.sqrt()` Method**: This method will return the square root of a number.
- **The `Math.cbrt()` Method**: This method will return the cube root of a number.
- **The `Math.abs()` Method**: This method will return the absolute value of a number.
- **The `Math.pow()` Method**: This method takes two numbers and raises the first to the power of the second.

Common Number Methods

- **`isNaN()`**: `NaN` stands for "Not-a-Number". It's a special value that represents an unrepresentable or undefined numerical result. The `isNaN()` function property is used to determine whether a value is `NaN` or not. `Number.isNaN()` provides a more reliable way to check for `NaN` values, especially in cases where type coercion might lead to unexpected results with the global `isNaN()` function.

```
console.log(isNaN(NaN));       // true
console.log(isNaN(undefined)); // true
console.log(isNaN({}));        // true

console.log(isNaN(true));      // false
console.log(isNaN(null));      // false
console.log(isNaN(37));        // false


console.log(Number.isNaN(NaN));        // true
console.log(Number.isNaN(Number.NaN)); // true
console.log(Number.isNaN(0 / 0));      // true

console.log(Number.isNaN("NaN"));      // false
console.log(Number.isNaN(undefined));  // false
```

- **The `parseFloat()` Method**: This method parses a string argument and returns a floating-point number. It's designed to extract a number from the beginning of a string, even if the string contains non-numeric characters later on.
- **The `parseInt()` Method**: This method parses a string argument and returns an integer. `parseInt()` stops parsing at the first non-digit it encounters. For floating-point numbers, it returns only the integer part. If it can't find a valid integer at the start of the string, it returns `NaN`.

- **The `toFixed()` Method**: This method is called on a number and takes one optional argument, which is the number of digits to appear after the decimal point. It returns a string representation of the number with the specified number of decimal places.

--assignment--

Review the JavaScript Math topics and concepts.

JavaScript Objects Review

Object Basics

- **Definition**: An object is a data structure that is made up of properties. A property consists of a key and a value. To access data from an object you can use either dot notation or bracket notation.

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York"
};

console.log(person.name);  // Alice
console.log(person["name"]); // Alice
```

To set a property of an existing object you can use either dot notation or bracket notation together with the assignment operator.

```
const person = {
  name: "Alice",
  age: 30
};

person.job = "Engineer"
person["hobby"] = "Knitting"
console.log(person);  // {name: 'Alice', age: 30, job: 'Engineer', hobby:
'Knitting'}
```

Removing Properties From an Object

- **`delete` Operator**: This operator is used to remove a property from an object.

```
const person = {
  name: "Alice",
  age: 30,
  job: "Engineer"
};

delete person.job;

console.log(person.job); // undefined
```

Checking if an Object has a Property

- `hasOwnProperty()` **Method**: This method returns a boolean indicating whether the object has the specified property as its own property.

```
const person = {
  name: "Alice",
  age: 30
};

console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("job")); // false
```

- `in` **Operator**: This operator will return `true` if the property exists in the object.

```
const person = {
  name: "Bob",
  age: 25
};

console.log("name" in person);  // true
```

Accessing Properties From Nested Objects

- **Accessing Data**: Accessing properties from nested objects involves using the dot notation or bracket notation, much like accessing properties from simple objects. However, you'll need to chain these accessors to drill down into the nested structure.

```
const person = {
  name: "Alice",
  age: 30,
  contact: {
    email: "alice@example.com",
    phone: {
      home: "123-456-7890",
      work: "098-765-4321"
    }
  }
};

console.log(person.contact.phone.work); // "098-765-4321"
```

Primitive and Non Primitive Data Types

- **Primitive Data Types**: These data types include numbers, strings, booleans, `null`, `undefined`, and symbols. These types are called "primitive" because they represent single values and are not objects. Primitive values are immutable, which means once they are created, their value cannot be changed.
- **Non Primitive Data Types**: In JavaScript, these are objects, which include regular objects, arrays, and functions. Unlike primitives, non-primitive types can hold multiple values as properties or elements.

Object Methods

- **Definition**: Object methods are functions that are associated with an object. They are defined as properties of an object and can access and manipulate the object's data. The `this` keyword inside the method refers to the object itself, enabling access to its properties.

```

```

```
const person = {
  name: "Bob",
  age: 30,
  sayHello: function() {
    return "Hello, my name is " + this.name;
  }
};

console.log(person.sayHello()); // "Hello, my name is Bob"
```

Object Constructor

- **Definition**: In JavaScript, a constructor is a special type of function used to create and initialize objects. It is invoked with the `new` keyword and can initialize properties and methods on the newly created object. The `Object()` constructor creates a new empty object.

```
new Object()
```

Working with the Optional Chaining Operator (`?.`)

- **Definition**: This operator lets you safely access object properties or call methods without worrying whether they exist.

```
const user = {
  name: "John",
  profile: {
    email: "john@example.com",
    address: {
      street: "123 Main St",
      city: "Somewhere"
    }
  }
};

console.log(user.profile?.address?.street); // "123 Main St"
console.log(user.profile?.phone?.number);   // undefined
```

Object Destructuring

- **Definition**: Object destructuring allows you to extract values from objects and assign them to variables in a more concise and readable way.

```
const person = { name: "Alice", age: 30, city: "New York" };

const { name, age } = person;

console.log(name); // Alice
console.log(age);  // 30
```

Working with JSON

- **Definition**: JSON stands for JavaScript Object Notation. It is a lightweight, text-based data format that is commonly used to exchange data between a server and a web application.

```
```

```
{
  "name": "Alice",
  "age": 30,
  "isStudent": false,
  "list of courses": ["Mathematics", "Physics", "Computer Science"]
}
```

- `JSON.stringify()`: This method is used to convert a JavaScript object into a JSON string. This is useful when you want to store or transmit data in a format that can be easily shared or transferred between systems.

```
const user = {
  name: "John",
  age: 30,
  isAdmin: true
};

const jsonString = JSON.stringify(user);
console.log(jsonString); // '{"name":"John","age":30,"isAdmin":true}'
```

- `JSON.parse()`: This method converts a JSON string back into a JavaScript object. This is useful when you retrieve JSON data from a web server or localStorage and you need to manipulate the data in your application.

```
const jsonString = '{"name":"John","age":30,"isAdmin":true}';
const userObject = JSON.parse(jsonString);

// result: { name: 'John', age: 30, isAdmin: true }
console.log(userObject);
```

--assignment--

Review the JavaScript Objects topics and concepts.

JavaScript Regular Expressions Review

Regular Expressions and Common Methods

- **Definition**: Regular Expressions, or Regex, are used to create a "pattern", which you can then use to check against a string, extract text, and more.

```
const regex = /freeCodeCamp/;
```

- `test()` **Method**: This method accepts a string, which is the string to test for matches against the regular expression. This method will return a boolean if the string matches the regex.

```
const regex = /freeCodeCamp/;
const test = regex.test("e");
console.log(test); // false
```

- `match()` **Method**: This method accepts a regular expression, although you can also pass a string which will be constructed into a regular expression. The `match` method returns the match array for the string.

```
const regex = /freeCodeCamp/;
const match = "freeCodeCamp".match(regex);
console.log(match); // ["freeCodeCamp"]
```

- `replace()` **Method**: This method accepts two arguments: the regular expression to match (or a string), and the string to replace the match with (or a function to run against each match).

```
const regex = /Jessica/;
const str = "Jessica is rly kewl";
const replaced = str.replace(regex, "freeCodeCamp");
console.log(replaced); // "freeCodeCamp is rly kewl"
```

- `replaceAll` **Method**: This method is used to replace all occurrences of a specified pattern with a new string. This method will throw an error if you give it a regular expression without the global modifier.

```
const text = "I hate JavaScript! I hate programming!";
const newText = text.replaceAll("hate", "love");
console.log(newText);  // "I love JavaScript! I love programming!"
```

- `matchAll` **Method**: This method is used to retrieve all matches of a given regular expression in a string, including capturing groups, and returns them as an iterator. An iterator is an object that allows you to go through (or "iterate over") a collection of items.

```
const str = "JavaScript, Python, JavaScript, Swift, JavaScript";
const regex = /JavaScript/g;

const iterator = str.matchAll(regex);

for (let match of iterator) {
  console.log(match[0]); // "JavaScript" for each match
}
```

Regular Expression Modifiers

- **Definition**: Modifiers, often referred to as "flags", modify the behavior of a regular expression.
- `i` **Flag**: This flag makes a regex ignore case.

```
const regex = /freeCodeCamp/i;
console.log(regex.test("freecodecamp")); // true
console.log(regex.test("FREECODECAMP")); // true
```

- `g` **Flag**: This flag, or global modifier, allows your regular expression to match a pattern more than once.

```
const regex = /freeCodeCamp/gi;
console.log(regex.test("freeCodeCamp")); // true
console.log(regex.test("freeCodeCamp is great")); // false
```

- **Anchor Definition**: The `^` anchor, at the beginning of the regular expression, says "match the start of the string". The `$` anchor, at the end of the regular expression, says "match the end of the string".

```
const start = /^freeCodeCamp/i;
const end = /freeCodeCamp$/i;
console.log(start.test("freecodecamp")); // true
console.log(end.test("freecodecamp")); // true
```

- `m` **Flag**: Anchors look for the beginning and end of the entire string. But you can make a regex handle multiple lines with the `m` flag, or the multi-line modifier.flag, or the multi-line modifier.

```
const start = /^freecodecamp/im;
const end = /freecodecamp$/im;
const str = `I love
freecodecamp
it's my favorite
`;
console.log(start.test(str)); // true
console.log(end.test(str)); // true
```

- `d` **Flag**: This flag expands the information you get in a match object.

```
const regex = /freecodecamp/di;
const string = "we love freecodecamp isn't freecodecamp great?";
console.log(string.match(regex));
```

- `u` **Flag**: This expands the functionality of a regular expression to allow it to match special unicode characters. The `u` flag gives you access to special classes like the `Extended_Pictographic` to match most emoji. There is also a `v` flag, which further expands the functionality of the unicode matching.
- `y` **Flag**: The sticky modifier behaves very similarly to the global modifier, but with a few exceptions. The biggest one is that a global regular expression will start from lastIndex and search the entire remainder of the string for another match, but a sticky regular expression will return null and reset the lastIndex to 0 if there is not immediately a match at the previous lastIndex.
- `s` **Flag**: The single-line modifier allows a wildcard character, represented by a `.` in regex, to match linebreaks - effectively treating the string as a single line of text.

Character Classes

- **Wildcard** `.`: Character classes are a special syntax you can use to match sets or subsets of characters. The first character class you should learn is the wild card class. The wild card is represented by a period, or dot, and matches ANY single character EXCEPT line breaks. To allow the wildcard class to match line breaks, remember that you would need to use the `s` flag.

```
const regex = /a./;
```

- `\d`: This will match all digits (`0-9`) in a string.

```
const regex = /\d/;
```

- `\w`: This is used to match any word character (`a-z0-9_`) in a string. A word character is defined as any letter, from a to z, or a number from 0 to 9, or the underscore character.

```
const regex = /\w/;
```

- `\s`: The white-space class `\s`, represented by a backslash followed by an `s`. This character class will match any white space, including new lines, spaces, tabs, and special unicode space characters.
- **Negating Special Character Classes**: To negate one of these character classes, instead of using a lowercase letter after the backslash, you can use the uppercase equivalent. The following example does not match a numerical character. Instead, it matches any single character that is NOT a numerical character.

```
const regex = /\D/;
```

- **Custom Character Classes**: You can create custom character classes by placing the character you wish match inside a set of square brackets.

```
const regex = /[abcdf]/;
```

Lookahead and Lookbehind Assertions

- **Definition**: Lookahead and lookbehind assertions allow you to match specific patterns based on the presence or lack of surrounding patterns.
- **Positive Lookahead Assertion**: This assertion will match a pattern when the pattern is followed by another pattern. To construct a positive lookahead, you need to start with the pattern you want to match. Then, use parentheses to wrap the pattern you want to use as your condition. After the opening parenthesis, use `?=` to define that pattern as a positive lookahead.

```
const regex = /free(?=code)/i;
```

- **Negative Lookahead Assertion**: This is a type of condition used in regular expressions to check that a certain pattern does not occur ahead in the string.

```
const regex = /free(?!code)/i;
```

- **Positive Lookbehind Assertion**: This assertion will match a pattern only if it is preceded by another specific pattern, without including the preceding pattern in the match.

```
const regex = /(?<=free)code/i;
```

- **Negative Lookbehind Assertion**: This assertion ensures that a pattern is not preceded by another specific pattern. It matches only if the specified pattern is not immediately preceded by the given sequence, without including the preceding sequence in the match.

```
const regex = /(?<!free)code/i;
```

Regex Quantifiers

- **Definition**: Quantifiers in regular expressions specify how many times a pattern (or part of a pattern) should appear. They help control the number of occurrences of characters or groups in a match. The following example is used to match the previous character exactly four times.

```
const regex = /^\d{4}$/;
```

- `*` : Matches 0 or more occurrences of the preceding element.
- `+`: Matches 1 or more occurrences of the preceding element.
- `?`: Matches 0 or 1 occurrence of the preceding element.
- `{n}`: Matches exactly n occurrences of the preceding element.
- `{n,}`: Matches n or more occurrences of the preceding element.
- `{n,m}`: Matches between n and m occurrences of the preceding element.

Capturing Groups and Backreferences

- **Capturing Groups**: A capturing group allows you to "capture" a portion of the matched string to use however you might need. Capturing groups are defined by parentheses containing the pattern to capture, with no leading characters like a lookahead.

```
const regex = /free(code)camp/i;
```

- **Non-Capturing Groups**: A non-capturing group is similar to a capturing group but does not store the matched portion of the string for later use. Non-capturing groups are defined by `(?:...)`.

```
const regex = /free(?:code)camp/i;
```

- **Backreferences**: A backreference in regular expressions refers to a way to reuse a part of the pattern that was matched earlier in the same expression. It allows you to refer to a captured group (a part of the pattern in parentheses) by its number. For example, `$1` refers to the first captured group.

```
const regex = /free(co+de)camp/i;
console.log("freecooooooooodecamp".replace(regex, "paid$1world"));
```

- You can use backreferences within the regex itself to match the same text captured by a previous group with a backslash and the capture group number. For example:

```
const regex = /(hello) \1/i;
console.log(regex.test("hello hello"));  // true
console.log(regex.test("hello world"));  // false
```

--assignment--

Review the JavaScript Regular Expressions topics and concepts.

JavaScript Strings Review

String Basics

- **Definition**: A string is a sequence of characters wrapped in either single quotes, double quotes or backticks. Strings are primitive data types and they are immutable. Immutability means that once a string is created, is cannot be changed.
- **Accessing Characters from a String**: To access a character from a string you can use bracket notation and pass in the index number. An index is the position of a character within a string, and it is zero-based.

```
const developer = "Jessica";
developer[0] // J
```

- `\n` **(Newline Character)**: You can create a newline in a string by using the `\n` newline character.

```
const poem = "Roses are red,\nViolets are blue,\nJavaScript is fun,\nAnd so
are you.";
console.log(poem);
```

- **Escaping Strings**: You can escape characters in a string by placing backlashes (`\`) in front of the quotes.

```
const statement = "She said, \"Hello!\"";
console.log(statement); // She said, "Hello!"
```

Template Literals (Template Strings) and String Interpolation

- **Definition**: Template literals are defined with backticks (`). They allow for easier string manipulation, including embedding variables directly inside a string, a feature known as string interpolation.

```
const name = "Jessica";
const greeting = `Hello, ${name}!`; // "Hello, Jessica!"
```

ASCII, the `charCodeAt()` Method and the `fromCharCode()` Method

- **ASCII**: ASCII, short for American Standard Code for Information Interchange, is a character encoding standard used in computers to represent text. It assigns a numeric value to each character, which is universally recognized by machines.
- **The `charCodeAt()` Method**: This method is called on a string and returns the ASCII code of the character at a specified index.

```
const letter = "A";
console.log(letter.charCodeAt(0));  // 65
```

- **The `fromCharCode()` Method**: This method converts an ASCII code into its corresponding character.

```
const char = String.fromCharCode(65);
console.log(char);  // A
```

Other Common String Methods

- **The `indexOf` Method**: This method is used to search for a substring within a string. If the substring is found, `indexOf` returns the index (or position) of the first occurrence of that substring. If the substring is not found, `indexOf` returns -1, which indicates that the search was unsuccessful.

```
const text = "The quick brown fox jumps over the lazy dog.";
console.log(text.indexOf("fox")); // 16
console.log(text.indexOf("cat")); // -1
```

- **The `includes()` Method**: This method is used to check if a string contains a specific substring. If the substring is found within the string, the method returns true. Otherwise, it returns false.

```
const text = "The quick brown fox jumps over the lazy dog.";
console.log(text.includes("fox")); // true
console.log(text.includes("cat")); // false
```

- **The `slice()` Method**: This method extracts a portion of a string and returns a new string, without modifying the original string. It takes two parameters: the starting index and the optional ending index.

```
const text = "freeCodeCamp";
console.log(text.slice(0, 4));  // "free"
console.log(text.slice(4, 8));  // "Code"
console.log(text.slice(8, 12)); // "Camp"
```

- **The `toUpperCase()` Method**: This method converts all the characters to uppercase letters and returns a new string with all uppercase characters.

```
const text = "Hello, world!";
console.log(text.toUpperCase()); // "HELLO, WORLD!"
```

- **The `toLowerCase()` Method**: This method converts all characters in a string to lowercase.

```
const text = "HELLO, WORLD!"
console.log(text.toLowerCase()); // "hello, world!"
```

- **The `replace()` Method**: This method allows you to find a specified value (like a word or character) in a string and replace it with another value. The method returns a new string with the replacement and leaves the original unchanged because JavaScript strings are immutable.

```
const text = "I like cats";
console.log(text.replace("cats", "dogs")); // "I like dogs"
```

- **The `repeat()` Method**: This method is used to repeat a string a specified number of times.

```
const text = "Hello";
console.log(text.repeat(3)); // "HelloHelloHello"
```

- **The `trim()` Method**: This method is used to remove whitespaces from both the beginning and the end of a string.

```
const text = "  Hello, world!  ";
console.log(text.trim()); // "Hello, world!"
```

- **The `trimStart()` Method**: This method removes whitespaces from the beginning (or "start") of the string.

```
const text = "  Hello, world!  ";
console.log(text.trimStart()); // "Hello, world!  "
```

- **The `trimEnd()` Method**: This method removes whitespaces from the end of the string.

```
const text = " Hello, world! ";
console.log(text.trimEnd()); // "  Hello, world!"
```

- **The `prompt()` Method**: This method of the `window` is used to get information from a user through the form of a dialog box. This method takes two arguments. The first argument is the message which will appear inside the dialog box, typically prompting the user to enter information. The second one is a default value which is optional and will fill the input field initially.

```
const answer = window.prompt("What's your favorite animal?"); // This will
change depending on what the user answers
```

--assignment--

Review the JavaScript Strings topics and concepts.

JavaScript Variables and Data Types Review

Working with HTML, CSS, and JavaScript

While HTML and CSS provide website structure, JavaScript brings interactivity to websites by enabling complex functionality, such as handling user input, animating elements, and even building full web applications.

Data Types in JavaScript

Data types help the program understand the kind of data it's working with, whether it's a number, text, or something else.

- **Number**: A number represents both integers and floating-point values. Examples of integers include 7, 19, and 90.
- **Floating point**: A floating point number is a number with a decimal point. Examples include 3.14, 0.5, and 0.0001.
- **String**: A string is a sequence of characters, or text, enclosed in quotes. `"I like coding"` and `'JavaScript is fun'` are examples of strings.
- **Boolean**: A boolean represents one of two possible values: `true` or `false`. You can use a boolean to represent a condition, such as `isLoggedIn = true`.
- **Undefined and Null**: An `undefined` value is a variable that has been declared but not assigned a value. A `null` value is an empty value, or a variable that has intentionally been assigned a value of `null`.

- **Object**: An object is a collection of key-value pairs. The key is the property name, and the value is the property value.

Here, the `pet` object has three properties or keys: `name`, `age`, and `type`. The values are `Fluffy`, `3`, and `dog`, respectively.

```
let pet = {
  name: "Fluffy",
  age: 3,
  type: "dog"
};
```

- **Symbol**: The Symbol data type is a unique and immutable value that may be used as an identifier for object properties.

In this example below, two symbols are created with the same description, but they are not equal.

```
const crypticKey1= Symbol("saltNpepper");
const crypticKey2= Symbol("saltNpepper");
console.log(crypticKey1 === crypticKey2); // false
```

- **BigInt**: When the number is too large for the `Number` data type, you can use the BigInt data type to represent integers of arbitrary length.

By adding an `n` to the end of the number, you can create a BigInt.

```
const veryBigNumber = 12345678901234567890123456789012345678890n;
```

Variables in JavaScript

- Variables can be declared using the `let` keyword.

```
let cityName;
```

- To assign a value to a variable, you can use the assignment operator `=`.

```
cityName = "New York";
```

- Variables declared using `let` can be reassigned a new value.

```
cityName = "Los Angeles";
console.log(cityName); // Los Angeles
```

- Apart from `let`, you can also use `const` to declare a variable. However, a `const` variable cannot be reassigned a new value.

```
const cityName = "New York";
cityName = "Los Angeles"; // TypeError: Assignment to constant variable.
```

- Variables declared using `const` find uses in declaring constants, that are not allowed to change throughout the code, such as `PI` or `MAX_SIZE`.

Variable Naming Conventions

- Variable names should be descriptive and meaningful.
- Variable names should be camelCase like `cityName`, `isLoggedIn`, and `veryBigNumber`.
- Variable names should not start with a number. They must begin with a letter, `_`, or `$`.
- Variable names should not contain spaces or special characters, except for `_` and `$`.
- Variable names should not be reserved keywords.
- Variable names are case-sensitive. `age` and `Age` are different variables.

Strings and String immutability in JavaScript

- Strings are sequences of characters enclosed in quotes. They can be created using single quotes and double quotes.

```
let correctWay = 'This is a string';
let alsoCorrect = "This is also a string";
```

- Strings are immutable in JavaScript. This means that once a string is created, you cannot change the characters in the string. However, you can still reassign strings to a new value.

```
let firstName = "John";
firstName = "Jane"; // Reassigning the string to a new value
```

String Concatenation in JavaScript

- Concatenation is the process of joining multiple strings or combining strings with variables that hold text. The `+` operator is one of the simplest and most frequently used methods to concatenate strings.

```
let studentName = "Asad";
let studentAge = 25;
let studentInfo = studentName + " is " + studentAge + " years old.";
console.log(studentInfo); // Asad is 25 years old.
```

- If you need to add or append to an existing string, then you can use the `+=` operator. This is helpful when you want to build upon a string by adding more text to it over time.

```
let message = "Welcome to programming, ";
message += "Asad!";
console.log(message); // Welcome to programming, Asad!
```

- Another way you can concatenate strings is to use the `concat()` method. This method joins two or more strings together.

```
let firstName = "John";
let lastName = "Doe";
let fullName = firstName.concat(" ", lastName);
console.log(fullName); // John Doe
```

Logging Messages with `console.log()`

- The `console.log()` method is used to log messages to the console. It's a helpful tool for debugging and testing your code.

```
console.log("Hello, World!");
// Output: Hello, World!
```

Semicolons in JavaScript

- Semicolons are primarily used to mark the end of a statement. This helps the JavaScript engine understand the separation of individual instructions, which is crucial for correct execution.

```
let message = "Hello, World!"; // first statement ends here
let number = 42; // second statement starts here
```

- Semicolons help prevent ambiguities in code execution and ensure that statements are correctly terminated.

Comments in JavaScript

- Any line of code that is commented out is ignored by the JavaScript engine. Comments are used to explain code, make notes, or temporarily disable code.
- Single-line comments are created using `//`.

```
// This is a single-line comment and will be ignored by the JavaScript engine
```

- Multi-line comments are created using `/*` to start the comment and `*/` to end the comment.

```
/*
This is a multi-line comment.
It can span multiple lines.
*/
```

JavaScript as a Dynamically Typed Language

- JavaScript is a dynamically typed language, which means that you don't have to specify the data type of a variable when you declare it. The JavaScript engine automatically determines the data type based on the value assigned to the variable.

```
let error = 404; // JavaScript treats error as a number
error = "Not Found"; // JavaScript now treats error as a string
```

- Other languages, like Java, that are not dynamically typed would result in an error:

```
int error = 404; // value must always be an integer
error = "Not Found"; // This would cause an error in Java
```

Using the `typeof` Operator

- The `typeof` operator is used to check the data type of a variable. It returns a string indicating the type of the variable.

```
let age = 25;
console.log(typeof age); // "number"

let isLoggedIn = true;
console.log(typeof isLoggedIn); // "boolean"
```

- However, there's a well-known quirk in JavaScript when it comes to `null`. The `typeof` operator returns `"object"` for `null` values.

```
let user = null;
console.log(typeof user); // "object"
```

--assignment--

Review the JavaScript Variables and Data Types topics and concepts.

JavaScript and Accessibility Review

Common ARIA Accessibility Attributes

- **`aria-expanded` attribute**: Used to convey the state of a toggle (or disclosure) feature to screen reader users.

```
<button aria-expanded="true">Menu</button>
```

- **`aria-haspopup` attribute**: This state is used to indicate that an interactive element will trigger a pop-up element when activated. You can only use the `aria-haspopup` attribute when the pop-up has one of the following roles: `menu`, `listbox`, `tree`, `grid`, or `dialog`. The value of `aria-haspopup` must be either one of these roles or `true`, which is the same as `menu`.

```
<button
  id="menubutton"
  aria-haspopup="menu"
  aria-controls="filemenu"
  aria-expanded="false"
>
    File
</button>

<ul
  id="filemenu"
  role="menu"
  aria-labelledby="menubutton"
  hidden
>
  <li role="menuitem" tabindex="-1">Open</li>
  <li role="menuitem" tabindex="-1">New</li>
  <li role="menuitem" tabindex="-1">Save</li>
  <li role="menuitem" tabindex="-1">Delete</li>
</ul>
```

- `aria-checked` **attribute**: This attribute is used to indicate whether an element is in the checked state. It is most commonly used when creating custom checkboxes, radio buttons, switches, and listboxes.

```
<div role="checkbox" aria-checked="true" tabindex="0">Checkbox</div>
```

- `aria-disabled` **attribute**: This state is used to indicate that an element is disabled only to people using assistive technologies, such as screen readers.

```
<div role="button" tabindex="-1" aria-disabled="true">Edit</div>
```

- `aria-selected` **attribute**: This state is used to indicate that an element is selected. You can use this state on custom controls like a tabbed interface, a listbox, or a grid.

```
<div role="tablist">
  <button role="tab" aria-selected="true">Tab 1</button>
  <button role="tab" aria-selected="false">Tab 2</button>
  <button role="tab" aria-selected="false">Tab 3</button>
</div>
```

- `aria-controls` **attribute**: Used to associate an element with another element that it controls. This helps people using assistive technologies understand the relationship between the elements.

```
<div role="tablist">
  <button
    role="tab"
    id="tab1"
    aria-controls="section1"
    aria-selected="true"
  >
    Tab 1
  </button>
  <button
    role="tab"
    id="tab2"
    aria-controls="section2"
    aria-selected="false"
  >
    Tab 2
  </button>
  <button
    role="tab"
    id="tab3"
    aria-controls="section3"
    aria-selected="false"
  >
    Tab 3
  </button>
</div>
```

- `hidden` **attribute**: Hides inactive panels from both visual and assistive technology users.

Working with Live Regions and Dynamic Content

- `aria-live` **attribute**: Makes part of a webpage a live region, meaning any updates inside that area will be announced by a screen reader so users don't miss important changes.
- `polite` **value**: Most live regions use this value. This value means that the update is not urgent, so the screen reader can wait until it finishes any current announcement or the user completes their current action before announcing the update.

Here is an example of a live region that is dynamically updated by JavaScript:

```
<div aria-live="polite" id="status"></div>
```

```
const statusEl = document.getElementById("status");
statusEl.textContent = "Your file has been successfully uploaded.";
```

- `contenteditable` **attribute**: Turns the element into a live editor, allowing users to update its content as if it were a text field. When there is no visible label or heading for a contenteditable region, add an accessible name using the `aria-label` attribute to help screen reader users understand the purpose of the editable area.

```
<div contenteditable="true" aria-label="Note editor">
  Editable content goes here
</div>
```

`focus` and `blur` Events

- `blur` **event**: Fires when an element loses focus.

```
element.addEventListener("blur", () => {
  // Handle when user leaves the element
});
```

- `focus` **event**: Fires when an element receives focus.

```
element.addEventListener("focus", () => {
  // Handle when user enters the element
});
```

--assignment--

Review the JavaScript and Accessibility topics and concepts.

Step 1

A teacher has finished grading their students' tests and needs your help to calculate the average score for the class.

Complete the `getAverage` function which takes in an array of test scores and returns the average score.

The average is calculated by adding up all the scores and dividing by the total number of scores.

```
average = sum of all scores / total number of scores
```

A couple of function calls have been provided for you so you can test out your code.

Tips

- You can use a loop to iterate over the `scores` array and add up all the scores.
- You can use the `length` property to get the total number of scores.

--hints--

Your `getAverage` function should return a number.

```
assert.strictEqual(typeof getAverage([92, 88, 12, 77, 57, 100, 67, 38, 97,
89]), 'number');
```

`getAverage([92, 88, 12, 77, 57, 100, 67, 38, 97, 89])` should return `71.7`.

```
assert.strictEqual(getAverage([92, 88, 12, 77, 57, 100, 67, 38, 97, 89]),
71.7);
```

`getAverage([45, 87, 98, 100, 86, 94, 67, 88, 94, 95])` should return `85.4`.

```
assert.strictEqual(getAverage([45, 87, 98, 100, 86, 94, 67, 88, 94, 95]),
85.4);
```

`getAverage([38, 99, 87, 100, 100, 100, 100, 100, 100, 100])` should return `92.4`.

```
assert.strictEqual(getAverage([38, 99, 87, 100, 100, 100, 100, 100, 100,
100]), 92.4);
```

Your `getAverage` function should return the average score of the test scores.

```
assert.strictEqual(getAverage([52, 56, 60, 65, 70, 75, 80, 85, 90, 95]),
72.8);
assert.strictEqual(getAverage([45, 50, 55, 60, 65, 70, 75, 80, 85, 90]),
67.5);
assert.strictEqual(getAverage([38, 42, 46, 50, 54, 58, 62, 66, 70, 74]), 56);
```

--seed--

--seed-contents--

```
--fcc-editable-region--
function getAverage(scores) {

}
```

```
console.log(getAverage([92, 88, 12, 77, 57, 100, 67, 38, 97, 89]));
console.log(getAverage([45, 87, 98, 100, 86, 94, 67, 88, 94, 95]));
--fcc-editable-region--
```

Step 2

Now the teacher needs your help converting the student score to a letter grade.

Complete the `getGrade` function that takes a number `score` as a parameter. Your function should return a string representing a letter grade based on the score.

Here are the scores and their corresponding letter grades:

Score Range	Grade
`100`	`"A++"`
`90 - 99`	`"A"`
`80 - 89`	`"B"`
`70 - 79`	`"C"`
`60 - 69`	`"D"`
`0 - 59`	`"F"`

Tips

- Remember that you learned about conditional statements (`if`, `else if`, and `else`).
- Remember that you learned about comparison operators (`>`, `<`, `>=`, `<=`, `===`).
- Remember that you learned about the logical AND operator (`&&`).

--hints--

Your `getGrade` function should return `"A++"` if the score is `100`.

```
assert.strictEqual(getGrade(100), "A++");
```

Your `getGrade` function should return `"A"` if the score is `94`.

```
assert.strictEqual(getGrade(94), "A");
```

Your `getGrade` function should return `"B"` if the score is between `80` and `89`.

```
assert.strictEqual(getGrade(80), "B");
assert.strictEqual(getGrade(83), "B");
assert.strictEqual(getGrade(88), "B");
```

Your `getGrade` function should return `"C"` if the score is `78`.

```
assert.strictEqual(getGrade(75), "C");
```

Your `getGrade` function should return `"D"` if the score is between `60` and `69`.

```
assert.strictEqual(getGrade(60), "D");
assert.strictEqual(getGrade(63), "D");
assert.strictEqual(getGrade(68), "D");
```

Your `getGrade` function should return `"F"` if the score is `57`.

```
assert.strictEqual(getGrade(57), "F");
```

Your `getGrade` function should return `"F"` if the score is between `0` and `59`.

```
assert.strictEqual(getGrade(0), "F");
assert.strictEqual(getGrade(30), "F");
assert.strictEqual(getGrade(43), "F");
```

--seed--

--seed-contents--

```
function getAverage(scores) {
  let sum = 0;

  for (const score of scores) {
    sum += score;
  }

  return sum / scores.length;
}
--fcc-editable-region--
function getGrade(score) {

}

console.log(getGrade(96));
console.log(getGrade(82));
console.log(getGrade(56));
--fcc-editable-region--
```

Step 3

The teacher is really happy with the program you have created so far. But now they want to have an easy way to check if a student has a passing grade. A passing grade is anything that is not an `"F"`.

Complete the function `hasPassingGrade` that takes a student score as a parameter. Your function should return `true` if the student has a passing grade and `false` if they do not.

Tips

- Use the `getGrade` function to get the student's grade. Then check if the grade is passing or not.

--hints--

Your `hasPassingGrade` function should return a boolean value.

```
assert.strictEqual(typeof hasPassingGrade(100), "boolean");
```

Your `hasPassingGrade` function should return `true` if the grade is an `"A"`.

```
assert.isTrue(hasPassingGrade(100));
```

Your `hasPassingGrade` function should return `false` if the grade is an `"F"`.

```
assert.isFalse(hasPassingGrade(53));
```

Your `hasPassingGrade` function should return `true` for all passing grades.

```
assert.isTrue(hasPassingGrade(87));
assert.isTrue(hasPassingGrade(60));
assert.isTrue(hasPassingGrade(73));
assert.isTrue(hasPassingGrade(96));
```

--seed--

--seed-contents--

```
function getAverage(scores) {
  let sum = 0;

  for (const score of scores) {
    sum += score;
  }

  return sum / scores.length;
}

function getGrade(score) {
  if (score === 100) {
    return "A++";
  } else if (score >= 90) {
    return "A";
  } else if (score >= 80) {
    return "B";
  } else if (score >= 70) {
    return "C";
  } else if (score >= 60) {
    return "D";
  } else {
    return "F";
  }
}

--fcc-editable-region--
function hasPassingGrade(score) {


```

```
}


console.log(hasPassingGrade(100));
console.log(hasPassingGrade(53));
console.log(hasPassingGrade(87));
--fcc-editable-region--
```

Step 4

Now that the teacher has all of the information they need, they want to be able to message the student with the results.

Complete the `studentMsg` function with `totalScores` and `studentScore` for parameters. The function should return a string representing a message to the student.

If the student passed the course, the string should follow this format:

```
Class average: average-goes-here. Your grade: grade-goes-here. You passed the course.
```

If the student failed the course, the string should follow this format:

```
Class average: average-goes-here. Your grade: grade-goes-here. You failed the course.
```

Replace `average-goes-here` with the average of the total scores. Replace `grade-goes-here` with the student's grade.

Tips

- Use the `getAverage` function to get the class average.
- Use the `getGrade` function to get the student's grade.
- Use string concatenation (`+`) to build the message.
- Be careful with the punctuation and spaces in the message.

--hints--

`studentMsg([92, 88, 12, 77, 57, 100, 67, 38, 97, 89], 37)` should return the following message: `"Class average: 71.7. Your grade: F. You failed the course."`.

```
assert.strictEqual(studentMsg([92, 88, 12, 77, 57, 100, 67, 38, 97, 89], 37), "Class average: 71.7. Your grade: F. You failed the course.");
```

`studentMsg([56, 23, 89, 42, 75, 11, 68, 34, 91, 19], 100)` should return the following message: `"Class average: 50.8. Your grade: A++. You passed the course."`.

```
assert.strictEqual(studentMsg([56, 23, 89, 42, 75, 11, 68, 34, 91, 19], 100), "Class average: 50.8. Your grade: A++. You passed the course.");
```

Your `studentMsg` function should return the correct message based on the student's score and the class average.

```
assert.strictEqual(studentMsg([33, 44, 55, 66, 77, 88, 99, 100], 92), "Class
average: 70.25. Your grade: A. You passed the course.");
assert.strictEqual(studentMsg([33, 44, 55, 66, 77, 88, 99, 100], 57), "Class
average: 70.25. Your grade: F. You failed the course.");
```

--seed--

--seed-contents--

```
function getAverage(scores) {
  let sum = 0;

  for (const score of scores) {
    sum += score;
  }

  return sum / scores.length;
}

function getGrade(score) {
  if (score === 100) {
    return "A++";
  } else if (score >= 90) {
    return "A";
  } else if (score >= 80) {
    return "B";
  } else if (score >= 70) {
    return "C";
  } else if (score >= 60) {
    return "D";
  } else {
    return "F";
  }
}

function hasPassingGrade(score) {
  return getGrade(score) !== "F";
}

--fcc-editable-region--
function studentMsg(totalScores, studentScore) {

}
console.log(studentMsg([92, 88, 12, 77, 57, 100, 67, 38, 97, 89], 37));
--fcc-editable-region--
```

--solutions--

```
function getAverage(scores) {
  let sum = 0;

  for (const score of scores) {
```

```
    sum += score;
  }

  return sum / scores.length;
}

function getGrade(score) {
  if (score === 100) {
    return "A++";
  } else if (score >= 90) {
    return "A";
  } else if (score >= 80) {
    return "B";
  } else if (score >= 70) {
    return "C";
  } else if (score >= 60) {
    return "D";
  } else {
    return "F";
  }
}

function hasPassingGrade(score) {
  return getGrade(score) !== "F";
}

function studentMsg(totalScores, studentScore) {
  let average = getAverage(totalScores);
  let grade = getGrade(studentScore);

  return `Class average: ${average}. Your grade: ${grade}. You ${
    hasPassingGrade(studentScore) ? "passed" : "failed"
  } the course.`;
}
```

Local Storage and CRUD Review

Persistent Storage

- **Definition**: Persistent storage refers to a way of saving data in a way that it stays available even after the power is turned off or the device is restarted.

Create, Read, Update, Delete (CRUD)

- **Create**: This refers to the process of creating new data. For example, in a web app, this could be when a user adds a new post to a blog.
- **Read**: This is the operation where data is retrieved from a database. For instance, when you visit a blog post or view your profile on a website, you're performing a read operation to fetch and display data stored in the database.
- **Update**: This involves modifying existing data in the database. An example would be editing a blog post or updating your profile information.
- **Delete**: This is the operation that removes data from a database. For instance, when you delete a blog post or account, you're performing a delete operation.

HTTP Methods

- **Definition**: HTTP stands for Hypertext Transfer Protocol and it is the foundation for data communication on the web. There are HTTP methods which define the actions that can be performed on resources over the web. The common methods are GET, POST, PUT, PATCH, DELETE.

- `GET` **Method**: This is used to fetch data from a server.
- `POST` **Method**: This is used to submit data to a server which creates a new resource.
- `PUT` **Method**: This is used to update a resource by replacing it entirely.
- `PATCH` **Method**: This is used to partially update a resource.
- `DELETE` **Method**: This is used to remove records from a database.

`localStorage` and `sessionStorage` Properties

- **Web Storage API**: This API provides a mechanism for browsers to store key-value pairs right within the browser, allowing developers to store information that can be used across different page reloads and sessions. The two main components for the Web Storage API are the `localStorage` and `sessionStorage` properties.
- `localStorage` **Property**: `localStorage` is the part of the Web Storage API that allows data to persist even after the browser window is closed or the page is refreshed. This data remains available until it is explicitly removed by the application or the user.
- `localStorage.setItem()` **Method**: This method is used to store a key-value pair in `localStorage`.

```
localStorage.setItem('username', 'Jessica');
```

- `localStorage.getItem()` **Method**: This method is used to retrieve the value of a given key from `localStorage`.

```
localStorage.setItem('username', 'codingRules');

let username = localStorage.getItem('username');
console.log(username); // codingRules
```

- `localStorage.removeItem()` **Method**: This method is used to remove a specific item from `localStorage` using its key.

```
localStorage.removeItem('username');
```

- `localStorage.clear()` **Method**: This method is used to clear all of the stored data in `localStorage`.

```
localStorage.clear();
```

- `sessionStorage` **Property**: Stores data that lasts only for the current session and is cleared when the browser tab or window is closed.
- `sessionStorage.setItem()` **Method**: This method is used to store a key-value pair in `sessionStorage`.

```
sessionStorage.setItem('cart', '3 items');
```

- `sessionStorage.getItem()` **Method**: This method is used to retrieve the value of a given key from `sessionStorage`.

```
sessionStorage.setItem('cart', '3 items');
```

```
let cart = sessionStorage.getItem('cart');
console.log(cart); // '3 items'
```

- **`sessionStorage.removeItem()` Method**: This method is used to remove a specific item from `sessionStorage` using its key.

```
sessionStorage.removeItem('cart');
```

- **`sessionStorage.clear()` Method**: This method is used to clear all data stored in `sessionStorage`.

```
sessionStorage.clear();
```

Working with Cookies

- **Definition**: Cookies, also known as web cookies or browser cookies, are small pieces of data that a server sends to a user's web browser. These cookies are stored on the user's device and sent back to the server with subsequent requests. Cookies are essential in helping web applications maintain state and remember user information, which is especially important since HTTP is a stateless protocol.
- **Session Cookies**: These cookies only last for the duration of the user's session on the website. Once the user closes the browser or tab, the session cookie is deleted. These cookies are typically used for tasks like keeping a user logged in during their visit.
- **Secure Cookies**: These cookies are only sent over HTTPS, ensuring that they cannot be intercepted by an attacker in transit.
- **HttpOnly Cookies**: These cookies cannot be accessed or modified by JavaScript running in the browser, making them more secure against cross-site scripting (XSS) attacks.
- **Set-Cookie Header**: When you visit a website, the server can send a Set-Cookie header in the HTTP response. This header tells your browser to save a cookie with specific information. For example, it might store a unique ID that helps the site recognize you the next time you visit. You can manually set a cookie in JavaScript using `document.cookie`:

```
document.cookie = "organization=freeCodeCamp; expires=Fri, 31 Dec 2021
23:59:59 GMT; path=/";
```

To delete a cookie, you can set its expiration date to a time in the past.

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/";
```

Cache API

- **Definition**: Caching is the process of storing copies of files in a temporary storage location, so that they can be accessed more quickly. The Cache API is used to store network requests and responses, making web applications work more efficiently and even function offline. It is part of the broader Service Worker API and is crucial for creating Progressive Web Apps (PWAs) that can work under unreliable or slow network conditions. The Cache API is a storage mechanism that stores Request and Response objects. When a request is made to a server, the application can store the response and later retrieve it from the cache instead of making a new network request. This reduces load times, saves bandwidth, and improves the overall user experience.
- **Cache Storage**: This is used to store key-value pairs of HTTP requests and their corresponding responses. This enables efficient retrieval of previously requested resources, reducing the need to

fetch them from the network on subsequent visits and improving performance.
- **Cache-Control**: Developers can define how long a cached resource should be kept, and if it should be revalidated or served directly from cache.
- **Offline Support**: By using the Cache API, you can create offline-first web applications. For example, a PWA can serve cached assets when the user is disconnected from the network.

Negative Patterns and Client Side Storage

- **Excessive Tracking**: This refers to the practice of collecting and storing an overabundance of user data in client-side storage (such as cookies, local storage, or session storage) without clear, informed consent or a legitimate need. This often involves tracking user behavior, preferences, and interactions across multiple sites or sessions, which can infringe on user privacy.
- **Browser Fingerprinting**: A technique used to track and identify individual users based on unique characteristics of their device and browser, rather than relying on cookies or other traditional tracking methods. Unlike cookies, which are stored locally on a user's device, fingerprinting involves collecting a range of information that can be used to create a distinctive "fingerprint" of a user's browser session.
- **Setting Passwords in LocalStorage**: This might seem like a more obvious negative pattern, but setting any sensitive data like passwords in local storage poses a security risk. Local Storage is not encrypted and can be accessed easily. So you should never store any type of sensitive data in there.

IndexedDB

- **Definition**: IndexedDB is used for storing structured data in the browser. This is built into modern web browsers, allowing web apps to store and fetch JavaScript objects efficiently.

Cache/Service Workers

- **Definition**: A Service Worker is a script that runs in the background which is separate from your web page. It can intercept network requests, access the cache, and make the web app work offline. This is a key component of Progressive Web Apps.

--assignment--

Review the Local Storage and CRUD topics and concepts.

Loops and Sequences Review

Python Lists

- **Introduction**: In Python, the list data type is an ordered sequence of elements that can be composed of strings, numbers or even other lists. Lists are mutable and zero based indexed.

```
cities = ['Los Angeles', 'London', 'Tokyo']
```

- **Accessing Elements in a List**: To access an element from the `cities` list, you can reference its index number in the sequence:

```
cities = ['Los Angeles', 'London', 'Tokyo']
cities[0] # Los Angeles
```

- **Accessing Elements Using Negative Indexing**: To access the last element of any list, you can use `-1` as the index number:

```
cities = ['Los Angeles', 'London', 'Tokyo']
cities[-1] # Tokyo
```

- Negative indexing is used to access elements starting from the end of the list instead of the beginning at index `0`.
- **Creating Lists Using the `list()` constructor**: Lists can also be created using the `list()` constructor. The `list()` constructor is used to convert an iterable into a list:

```
developer = 'Jessica'

print(list(developer))
# Result: ['J', 'e', 's', 's', 'i', 'c', 'a']
```

- **Finding the Length of a List**: You can use the `len()` function to get the length of a list:

```
numbers = [1, 2, 3, 4, 5]
len(numbers) # 5
```

- **List Mutability**: Lists are mutable, meaning you can update any element in the list as long as you pass in a valid index number. To update lists at a particular index, you can assign a new value to that index:

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']
programming_languages[0] = 'JavaScript'
print(programming_languages) # ['JavaScript', 'Java', 'C++', 'Rust']
```

- **Index Out of Range Error**: If you pass in an index (either positive or negative) that is out of bounds for the list, then you will receive an `IndexError`:

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']
programming_languages[10] = 'JavaScript'

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
"""
```

- **Removing Elements from a List**: Elements can be removed from a list using the `del` keyword:

```
developer = ['Jane Doe', 23, 'Python Developer']
del developer[1]
print(developer) # ['Jane Doe', 'Python Developer']
```

- **Checking if an Element Exists in a List**: The `in` keyword can be used to check if an element exists in a list:

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']
```

```
'Rust' in programming_languages # True
'JavaScript' in programming_languages # False
```

- **Nesting Lists**: Lists can be nested inside other lists:

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
```

- To access the nested list, you will need to access it using index `2` since lists are zero-based indexed.

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
developer[2] # ['Python', 'Rust', 'C++']
```

- To further access the second language from that nested list, you will need to access it using index `1`:

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
developer[2][1] # Rust
```

- **Unpacking Values from a List:** Unpacking values from a list is a technique used to assign values from a list to new variables. Here is an example to unpack the `developer` list into new variables called `name`, `age` and `job` like this:

```
developer = ['Alice', 34, 'Rust Developer']
name, age, job = developer
```

- **Collecting Remaining Items From a List**: To collect any remaining elements from a list, you can use the asterisk (`*`) operator like this:

```
developer = ['Alice', 34, 'Rust Developer']
name, *rest = developer
```

- If the number of variables on the left side of the assignment operator doesn't match the total number of items in the list, then you will receive a `ValueError`.

- **Slicing Lists**: Slicing is the concept of accessing a portion of a list by using the slice operator `:`. To slice a list that starts at index `1` and ends at index `3`, you can use the following syntax:

```
desserts = ['Cake', 'Cookies', 'Ice Cream', 'Pie']
desserts[1:3] # ['Cookies', 'Ice Cream']
```

- **Step Intervals**: It is also possible to specify a step interval which determines how much to increment between the indices. Here is an example if you want to extract a list of just even numbers using slicing:

```
numbers = [1, 2, 3, 4, 5, 6]
numbers[1::2] # [2, 4, 6]
```

List Methods

- **append()**: Used to add an item to the end of the list. Here is an example of using the `append()` method to add the number `6` to this `numbers` list:

```
numbers = [1, 2, 3, 4, 5]
numbers.append(6)
print(numbers) # [1, 2, 3, 4, 5, 6]
```

- **Appending lists**: The `append()` method can also be used to add one list at the end of another:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [6, 8, 10]

numbers.append(even_numbers)
print(numbers) # [1, 2, 3, 4, 5, [6, 8, 10]]
```

- **extend()**: Used to add multiple items to the end of a list. Here is an example of adding the numbers `6`, `8`, and `10` to the end of the `numbers` list:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [6, 8, 10]

numbers.extend(even_numbers)
print(numbers) # [1, 2, 3, 4, 5, 6, 8, 10]
```

- **insert()**: Used to insert an item at a specific index in the list. Here is an example of using the `insert()` method:

```
numbers = [1, 2, 3, 4, 5]
numbers.insert(2, 2.5)

print(numbers) # [1, 2, 2.5, 3, 4, 5]
```

- **remove():** Used to remove an item from the list. The `remove()` method will only remove the first occurrence of an item in the list:

```
numbers = [1, 2, 3, 4, 5, 5, 5]
numbers.remove(5)

print(numbers) # [1, 2, 3, 4, 5, 5]
```

- **pop()**: Used to remove a specific item from the list and return it:

```
numbers = [1, 2, 3, 4, 5]
numbers.pop(1) # The number 2 is returned
```

- If you don't specify an element for the `pop` method, then the last element is removed.

```
numbers = [1, 2, 3, 4, 5]
numbers.pop() # The number 5 is returned
```

- **clear()**: Used to remove all items from the list:

```
numbers = [1, 2, 3, 4, 5]
numbers.clear()

print(numbers) # []
```

- **sort()**: The `sort()` method is used to sort the elements in place. Here is an example of sorting a random list of `numbers` in place:

```
numbers = [19, 2, 35, 1, 67, 41]
numbers.sort()

print(numbers) # [1, 2, 19, 35, 41, 67]
```

- **sorted()**: Used to sort the elements in a list and return a new sorted list instead of modifying the original list.
- **reverse()**: Used to reverse the order of the elements in a list:

```
numbers = [6, 5, 4, 3, 2, 1]
numbers.reverse()

print(numbers) # [1, 2, 3, 4, 5, 6]
```

- **index()**: Used to find the first index where an element can be found in a list:

```
programming_languages = ['Rust', 'Java', 'Python', 'C++']
programming_languages.index('Java') # 1
```

- If the element cannot be found using the `index()` method, then the result will be a `ValueError`.

Tuples in Python

- **Definition**: A tuple is a Python data type used to create an ordered sequence of values. Tuples can contain a mixed set of data types:

```
developer = ('Alice', 34, 'Rust Developer')
```

- Tuples are immutable, meaning the elements in the tuple cannot be changed once created. If you try to update one of the items in the tuple, you will get a `TypeError`:

```
programming_languages = ('Python', 'Java', 'C++', 'Rust')
programming_languages[0] = 'JavaScript'

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: "tuple" object does not support item assignment
"""
```

- **Accessing Elements from a Tuple**: To access an element from a tuple, use bracket notation and the index number:

```
developer = ('Alice', 34, 'Rust Developer')
developer[1] # 34
```

- Negative indexing can be used to access elements starting from the end of the tuple:

```
numbers = (1, 2, 3, 4, 5)
numbers[-2] # 4
```

- If you try to pass in an index number that exceeds or equals the length of the tuple, then you will receive an `IndexError`:

```
numbers = (1, 2, 3, 4, 5)
numbers[7]

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
"""
```

- A tuple can also be created using the `tuple()` constructor. Within the constructor, you can pass in different iterables like strings, lists and even other tuples.

```
developer = 'Jessica'

print(tuple(developer))
# Result: ('J', 'e', 's', 's', 'i', 'c', 'a')
```

- **Verifying Items in a Tuple**: To check if an item is in a tuple, you can use the `in` keyword like this:

```
programming_languages = ('Python', 'Java', 'C++', 'Rust')

'Rust' in programming_languages # True
'JavaScript' in programming_languages # False
```

- **Unpacking Tuples**: Items can be unpacked from a tuple like this:

```
developer = ('Alice', 34, 'Rust Developer')
name, age, job = developer
```

- If you need to collect any remaining elements from a tuple, you can use the asterisk (`*`) operator like this:

```
developer = ('Alice', 34, 'Rust Developer')
name, *rest = developer
```

- **Slicing Tuples**: Slicing can be used to extract a portion of a tuple. For example, the items `pie` and `cookies` can be sliced into a separate tuple:

```
desserts = ('cake', 'pie', 'cookies', 'ice cream')
desserts[1:3] # ('pie', 'cookies')
```

- **Removing Items from Tuples**: Removing an item from a tuple will raise a `TypeError` as tuples are immutable:

```
developer = ('Jane Doe', 23, 'Python Developer')
del developer[1]

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: "tuple" object doesn't support item deletion
"""
```

- **When to use a Tuple vs a List?**: If you need a dynamic collection of elements where you can add, remove and update elements, then you should use a list. If you know that you are working with a fixed and immutable collection of data, then you should use a tuple.

Common Tuple Methods

- **`count()`**: Used to determine how many times an item appears in a tuple. For example, you can check how many times the language `'Rust'` appears in the tuple:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.count('Rust') # 2
```

- If the specified item in the `count()` function is not present at all in the tuple, then the return value will be `0`:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.count('JavaScript') # 0
```

- If no arguments are passed to the `count()` function, then Python will return a `TypeError`.

- **index()**: Used to find the index where a particular item is present in the tuple. Here is an example of using the `index()` method to find the index for the language `'Java'`:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.index('Java') # 1
```

- If the specified item cannot be found, then Python will return a `ValueError`.

- You can pass an optional start index to the `index()` method to specify where to start searching for the item in the tuple:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')
programming_languages.index('Python', 3) # 5
```

- You can also pass in an optional end index to the `index()` method to specify where to stop searching for the item in the tuple:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python',
'JavaScript', 'Python')
programming_languages.index('Python', 2, 5) # 2
```

- **`sorted()`**: Used to sort the elements in any iterable and return a new sorted list. Here is an example of creating a new list of numbers using the `sorted()` function:

```
numbers = (13, 2, 78, 3, 45, 67, 18, 7)
sorted(numbers) # [2, 3, 7, 13, 18, 45, 67, 78]
```

- **Modifying Sorting Behavior**: You can customize the sorting behavior for an iterable using the optional `reverse` and `key` arguments. Here is an example of using the `key` argument to sort items in a tuple by length:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')
sorted(programming_languages, key=len)

# Result
# ['C++', 'Rust', 'Java', 'Rust', 'Python', 'Python']
```

- You can create a new list of values in reverse order, using the `reverse` argument like this:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')

print(sorted(programming_languages, reverse=True))

# Result
# ['Rust', 'Rust', 'Python', 'Python', 'Java', 'C++']
```

Loops in Python

- **Definition**: Loops are used to repeat a block of code for a set number of times.

- **`for` loop**: Used to iterate over a sequence (like a list, tuple or string) and execute a block of code for each item in that sequence. Here is an example of using a `for` loop to iterate through a list and print each language to the console:

```
programming_languages = ['Rust', 'Java', 'Python', 'C++']

for language in programming_languages:
    print(language)

"""
Result

Rust
Java
Python
C++
"""
```

- Here is an example of using a `for` loop to loop through the string `code` and print out each character:

```
for char in 'code':
    print(char)

"""
Result

c
o
d
e
"""
```

- `for` loops can be nested. Here is an example of using a nested `for` loop:

```
categories = ['Fruit', 'Vegetable']
foods = ['Apple', 'Carrot', 'Banana']

for category in categories:
    for food in foods:
        print(category, food)

"""
Result

Fruit Apple
Fruit Carrot
Fruit Banana
Vegetable Apple
Vegetable Carrot
Vegetable Banana
"""
```

- **`while` loop**: Repeats a block of code until the condition is `False`. Here is an example of using a `while` loop for a guessing game:

```
secret_number = 3
guess = 0

while guess != secret_number:
    guess = int(input('Guess the number (1-5): '))
    if guess != secret_number:
        print('Wrong! Try again.')

print('You got it!')

"""
Result

Guess the number (1-5): 2
Wrong! Try again.
Guess the number (1-5): 1
Wrong! Try again.
Guess the number (1-5): 3
You got it!
"""
```

- **`break` and `continue` statements**: Used in loops to modify the execution of a loop.
- The `break` statement is used to exit the loop immediately when a certain condition is met. Here is an example of using the `break` statement for a list of `developer_names`:

```
developer_names = ['Jess', 'Naomi', 'Tom']

for developer in developer_names:
    if developer == 'Naomi':
        break
    print(developer)
```

- The `continue` statement is used to skip that current iteration and move onto the next iteration of the loop. Here is an example to use the `continue` statement instead of a `break` statement:

```
developer_names = ['Jess', 'Naomi', 'Tom']

for developer in developer_names:
    if developer == 'Naomi':
        continue
    print(developer)
```

- Both `for` and `while` loops can be combined with an `else` clause, which is executed only when the loop was not terminated by a `break`:

```
words = ['sky', 'apple', 'rhythm', 'fly', 'orange']

for word in words:
    for letter in word:
        if letter.lower() in 'aeiou':
            print(f"'{word}' contains the vowel '{letter}'")
            break
    else:
        print(f"'{word}' has no vowels")
```

Ranges and Their Use in Loops

- **The `range()` function**: Used to generate a sequence of integers.

```
range(start, stop, step)
```

- The required `stop` argument is an integer(non-inclusive) that represents the end point for the sequence of numbers being generated. Here is an example of using the `range()` function:

```
for num in range(3):
    print(num)
```

- If a `start` argument is not specified, then the default will be `0`. By default the sequence of integers will increment by `1`. You can use the optional `step` argument to change the default increment value. Here is an example of generating a sequence of even integers from 2 up to but not including 11 (i.e., includes 10)

```
for num in range(2, 11, 2):
    print(num)
```

- If you don't provide any arguments to the `range()` function, then you will get a `TypeError`.
- The `range()` function only accepts integers for arguments and not floats. Using floats will also result in a `TypeError`:

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

- You can use a negative integer for the `step` argument to generate a sequence of integers in decrementing order:

```
for num in range(40, 0, -10):
    print(num)
```

- The `range()` function can also be used to create a list of integers by using it with the `list` constructor. The `list` constructor is used to convert an iterable into a list. Here is an example of generating a list of even integers between 2 and 10 inclusive:

```
numbers = list(range(2, 11, 2))
print(numbers) # [2, 4, 6, 8, 10]
```

`enumerate()` and `zip()` functions in Python

- **`enumerate()`**: used to iterate over a sequence and keep track of the index for each item in that sequence. The `enumerate()` function takes an iterable as an argument and returns an `enumerate` object that consist of the index and value of each item in the iterable.

```
languages = ['Spanish', 'English', 'Russian', 'Chinese']

for index, language in enumerate(languages):
    print(f'Index {index} and language {language}')

# Result
# Index 0 and language Spanish
# Index 1 and language English
# Index 2 and language Russian
# Index 3 and language Chinese
```

- The `enumerate()` function can also be used outside of a `for` loop:

```
languages = ['Spanish', 'English', 'Russian', 'Chinese']

print(list(enumerate(languages)))
```

- The `enumerate()` function also accepts an optional `start` argument that specifies the starting value for the count. If this argument is omitted, then the count will begin at `0`.

- **`zip()`** : Used to iterate over multiple iterables in parallel. Here's an example using the `zip()` function to iterate over `developers` and `ids`:

```
developers = ['Naomi', 'Dario', 'Jessica', 'Tom']
ids = [1, 2, 3, 4]

for name, id in zip(developers, ids):
    print(f'Name: {name}')
    print(f'ID: {id}')


"""
Result

Name: Naomi
ID: 1
Name: Dario
ID: 2
Name: Jessica
ID: 3
Name: Tom
ID: 4
"""
```

List comprehensions in Python

- **Definition**: List comprehension allows you to create a new list in a single line by combining the loop and the condition directly within square brackets. This makes the code shorter and often easier to read.

```
even_numbers = [num for num in range(21) if num % 2 == 0]
print(even_numbers)
```

Iterable methods

- **`filter()`**: Used to filter elements from an iterable based on a condition. It returns an iterator that contains only the elements that satisfy the condition. Here is an example of creating a new list of just words longer than four characters:

```
words = ['tree', 'sky', 'mountain', 'river', 'cloud', 'sun']

def is_long_word(word):
    return len(word) > 4

long_words = list(filter(is_long_word, words))
print(long_words) # ['mountain', 'river', 'cloud']
```

- **`map()`**: Used to apply a function to each item in an iterable and return a new iterable with the results. Here is an example of using the `map()` function to convert a list of celsius temperatures to fahrenheit:

```
```

```
celsius = [0, 10, 20, 30, 40]

def to_fahrenheit(temp):
    return (temp * 9/5) + 32

fahrenheit = list(map(to_fahrenheit, celsius))
print(fahrenheit) # [32.0, 50.0, 68.0, 86.0, 104.0]
```

- **`sum()`**: Used to get the sum from an iterable like a list or tuple. Here is an example of using the `sum()` function:

```
numbers = [5, 10, 15, 20]
total = sum(numbers)
print(total) # Result: 50
```

- You can also pass in an optional `start` argument which sets the initial value for the summation. Here is an updated example using the `start` argument as a positional argument:

```
numbers = [5, 10, 15, 20]
total = sum(numbers, 10) # positional argument
print(total) # 60
```

- You can also choose to use the `start` argument as a keyword argument like this instead:

```
numbers = [5, 10, 15, 20]
total = sum(numbers, start=10) # keyword argument
print(total) # 60
```

Lambda functions

- **Definition**: A lambda function in Python is a concise way to create a function without a name (an anonymous function).
- Lambda functions are often used as an argument to another function. Here is an example of a lambda function:

```
numbers = [1, 2, 3, 4, 5]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]
```

- Best practices for using lambda functions include not assigning them to a variable, keeping them simple and readable, and using them for short, one-off functions.

--assignment--

Review the Loops and Sequences topics and concepts.

Object Oriented Programming Review

What is Object-Oriented Programming?

- **Object-oriented programming**: A programming style in which developers treat everything in their code like a real-world object. It is popularly called OOP. The four key principles that help you organize and manage code effectively are **encapsulation**, **inheritance**, **polymorphism**, and **abstraction**
- **Classes**: The blueprint for creating objects. Every single object created from a class has attributes that define data and methods that determine the behaviors of the objects.

What is Encapsulation?

- **Encapsulation**: The bundling of the attributes and methods of an object into a single unit. It lets you hide the internal state of the object behind a simple set of public methods and attributes that act like doors. Behind those doors are private attributes and methods that control how the data changes and who can see it.
- **Example of Encapsulation**: If you want to track a wallet balance, you will allow deposit and withdrawal, but you won't want anyone to tamper with the wallet balance itself:

```
class Wallet:
   def __init__(self, balance):
       self.__balance = balance # Private attribute

   def deposit(self, amount):
       if amount > 0:
           self.__balance += amount # Add to the balance safely

   def withdraw(self, amount):
       if 0 < amount <= self.__balance:
           self.__balance -= amount # Remove from the balance safely

account = Wallet(500)
print(account.__balance) # AttributeError: 'Wallet' object has no attribute
'__balance'
```

- **Difference Between Prefixing Attributes with Single and Double Underscore**: Prefixing attributes and methods with a single underscore means they are meant for internal use. This is a convention, and it doesn't enforce accessing attributes from the outside. Prefixing attributes and methods with a double underscore effectively prevents them from being accessed from outside of their class.

What Are Getters and Setters?

- **Getters and Setters**: Methods that let you control how the attributes of a class are accessed and modified. You retrieve values with getters and you set values with setters.
- **Properties**: They connect getters and setters, and allow access to data. They run extra logic behind the scenes when you get, set, or delete values.
- **Why Properties Instead of Methods**: Properties are used instead of methods for better readability and cleaner code. They let you access values with dot notation, like regular attributes, without parentheses.
- **Creating a Getter**: To create a getter, you use the `@property` decorator. Here's a getter that gets the radius of a circle:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self): # A getter to get the radius
        return self._radius

    @property
```

```
    def area(self):  # A getter to calculate area
        return 3.14 * (self._radius ** 2)

my_circle = Circle(3)

print(my_circle.radius) # 3
print(my_circle.area) # 28.26
```

- **Creating a Setter**: To create the setter that will set the radius, you have to define another method with the same name and use `@<property_name>.setter` above it:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):  # A getter to get the radius
        return self._radius

    @radius.setter
    def radius(self, value):  # A setter to set the radius
        if value <= 0:
            raise ValueError('Radius must be positive')
        self._radius = value

my_circle = Circle(3)
print('Initial radius:', my_circle.radius) # Initial radius: 3

my_circle.radius = 8
print('After modifying the radius:', my_circle.radius) # After modifying the
radius: 8
```

- **How Python Handles Getters and Setters**: Once you define getters and setters, Python automatically calls them under the hood whenever you use normal attribute syntax this way:

```
my_circle.radius # This will call the getter
my_circle.radius = 4 # This will call the setter
```

When setting a value, you should not assign to the property name itself because that will cause a `RecursionError`. Use a separate internal name, often with an underscore, to store the value.

- **Deleter**: After setting and getting a value with setter and getter, you can control how it is deleted with a `deleter`. A deleter runs custom logic when you use the `del` statement on a property. To create a deleter, you use the `@<property_name>.deleter` decorator.

```
  # Deleter
    @radius.deleter
    def radius(self):
        print("Deleting radius...")
        del self._radius
```

What Is Inheritance and How Does It Promote Code Reuse?

- **Inheritance**: The process by which a child class uses the attributes and methods of a parent class. Inheritance promotes code reuse, provides clear hierarchies, and customizes behavior without rewriting everything. To implement inheritance, a child class takes in the name of a parent class:

```
class Parent:
    # Parent attributes and methods

class Child(Parent):
    # Child inherits, extends, and/or overrides where necessary
```

- **Single and Multiple Inheritance**: When a child class inherits properties and methods from a single parent, as you can see above, the process is called **single inheritance**. When a child class inherits properties and methods from more than one parent, that is **multiple inheritance**. Here's the syntax for that:

```
class Parent:
    # Attributes and methods for Parent

class Child:
    # Attributes and methods for Child

class GrandChild(Parent, Child):
    # GrandChild inherits from both Parent and Child
    # GrandChild can combine or override behavior from each
```

- `super()` **Function**: A function that lets you override a method from a parent inside a child class.

What Is Polymorphism and How Does It Promote Code Reuse?

- **Polymorphism**: The OOP principle that lets different classes use the same method name, but each class implements it differently when called. Here's the syntax for it:

```
class A:
   def action(self): ...

class B:
   def action(self): ...

class C:
   def action(self): ...

Class().method()  # Works for A, B, or C
```

- **Inheritance-based polymorphism**: A parent sets up a method, and each child class twists it to their use.

What is Name Mangling and How Does it Work?

- **Name Mangling**: A process in which Python internally renames an attribute prefixed with a double underscore by adding an underscore and the class name as a prefix, turning `__attribute` into `_ClassName__attribute`.
- **The Purpose of Name Mangling**: The main purpose of name mangling is to prevent accidental attribute and method overriding when you use inheritance. Here's a code that makes that more understandable:

```
class Parent:
    def __init__(self):
        self.__data = 'Parent data'

class Child(Parent):
```

```
    def __init__(self):
        super().__init__()
        self.__data = 'Child data'

c = Child()
print(c.__dict__) # {'_Parent__data': 'Parent data', '_Child__data': 'Child
data'}
```

What Is Abstraction and How Does It Help Keep Complex Systems Organized?

- **Abstraction**: A programming concept in which complex implementation details of object or system are hidden and only the essential features are shown. In Python and other programming languages, abstraction simplifies complex systems by increasing reusability.
- **Example of Abstraction**: A good example of abstraction in everyday life is a car letting you just use the wheel, pedals, and shifter without knowing how the engine or brakes work.
- **How Python Implements Abstraction**: Python implements abstraction through the `abc` module. The module provides the `ABC` class (abstract base class) and the `@abstractmethod` decorator. An abstract base class (ABC) defines the common methods and properties subclasses must implement. It can't be instantiated.
- **How Abstract Method is Defined**: An abstract method is defined with `@abstractmethod` and must be overridden in subclasses, even if it has a default implementation. The basic syntax of abstraction looks like this:

```
from abc import ABC, abstractmethod

# Define an abstract base class
class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

# Concrete subclass that implements the abstract method
class ConcreteClassOne(AbstractClass):
    def abstract_method(self):
        print('Implementation in ConcreteClassOne')

# Another concrete subclass
class ConcreteClassTwo(AbstractClass):
    def abstract_method(self):
        print('Implementation in ConcreteClassTwo')
```

--assignment--

Review the Object Oriented Programming topics and concepts.

Python Review

Review the concepts below to prepare for the upcoming exam.

First topic

Describe

Second topic

Describe

--assignment--

Review the Python topics and concepts.

Python Basics Review

What is Python?

- **Introduction**: Python is a general-purpose programming language known for its simplicity and ease of use. Python is used in many fields like data science and machine learning, web development, scripting and automation, embedded systems and IoT, and much more.
- **Common Use Cases**: Python is used in data science, machine learning, web development, cybersecurity, automation and microcomputers like the Raspberry Pi and MicroPython-compatible boards.

Python in Your Local Environment

- **Installation**: The best way to install Python on Windows, Mac, and Linux is to download the installer from the official Python website (`https://www.python.org/`).

Variables

- **Declaring Variables**: To declare a variable, you start with the variable name followed by the assignment operator (`=`) and then the data type. This can be a number, string, boolean, etc. Here are some examples:

```
name = 'John Doe'
age = 25
```

- **Naming Conventions for Variables**: Here are the naming conventions you should use for variables:
    - Variable names can only start with a letter or an underscore (_), not a number.
    - Variable names can only contain alphanumeric characters (a-z, A-Z, 0-9) and underscores (_).
    - Variable names are case-sensitive — `age`, `Age`, and `AGE` are all considered unique.
    - Variable names cannot be one of Python's reserved keywords such as `if`, `class`, or `def`.
    - Variables names with multiple words are separated by underscores. Ex. `snake_case`.

Comments

- **Single Line Comments**: These types of comments should be used for short notes you wish to leave in your code.

```
# This is a single line comment
```

- **Multi-line Strings**: These types of strings can be used to leave larger notes or to comment out sections of code

```
"""
This is is a multi-line string.
Here is some code commented out.
```

```
name = 'John Doe'
age = 25
"""
```

- **`print()` Function**: To print data to the console, you can use the `print()` function like this:

```
print('Hello world!') # Hello world!
```

Common Data Types in Python

- **Introduction**: Python is a dynamically-typed language like JavaScript, meaning you don't need to explicitly declare types for variables. The language knows what the type of a variable is based on what you assign to the variable.
- **Integer**: A whole number without decimals:

```
my_integer_var = 10
print('Integer:', my_integer_var) # Integer: 10
```

- **Float**: A number with decimals:

```
my_float_var = 4.50
print('Float:', my_float_var) # Float 4.50
```

- **Complex**: A number with a real and imaginary part:

```
my_complex_var = 3 + 4j
print('Complex:', my_complex_var) # Complex: (3+4j)
```

- **String**: A sequence of characters wrapped in quotes:

```
my_string_var = 'hello'
print('String:', my_string_var) # String: hello
```

- **Boolean**: A value representing either `True` or `False`:

```
my_boolean_var = True
print('Boolean:', my_boolean_var) # Boolean: True
```

- **Set**: An unordered collection of unique elements:

```
my_set_var = {7, 5, 8}
print('Set:', my_set_var) # Set: {7, 5, 8}
```

- **Dictionary**: A collection of key-value pairs, enclosed in curly braces:

```
my_dictionary_var = {"name": "Alice", "age": 25}
print('Dictionary:', my_dictionary_var) # Dictionary: {'name': 'Alice',
```

```
'age': 25}
```

- **Tuple**: An immutable ordered collection, enclosed in parentheses:

```
my_tuple_var = (7, 5, 8)
print('Tuple:', my_tuple_var) # Tuple: (7, 5, 8)
```

- **Range**: A sequence of numbers, often used in loops:

```
my_range_var = range(5)
print(my_range_var) # range(0, 5)
```

- **List**: An ordered collection of elements that supports different data types:

```
my_list = [22, 'Hello world', 3.14, True]
print(my_list) # [22, 'Hello world', 3.14, True]
```

- **None**: A special value that represents the absence of a value:

```
my_none_var = None
print('None:', my_none_var) # None: None
```

Immutable and Mutable Types

- **Immutable Types**: These types cannot change once declared, although you can point their variables at something new, which is called reassignment. They include integer, float, complex, boolean, string, tuple, range, and `None`.
- **Mutable Types**: These types can change once declared. You can add, remove, or update their items. They include collection types such as list, set, and dictionary.
- **`type()` Function**: To see the type for a variable, you can use the `type()` function like this:

```
greeting = 'Hello there!'
age = 21

print(type(greeting)) # <class 'str'>
print(type(age)) # <class 'int'>
```

- **`isinstance()` Function**: This is used to check if a variable matches a specific data type:

```
print(isinstance('Hello world', str)) # True
print(isinstance('John Doe', int)) # False
```

Working with Strings

- **Definition**: As you recall from JavaScript, strings are immutable which means you can not change them after they have been created. In Python, you can use either single or double quotes. It is recommended to chose a rule and stick with it:

```
```

```
developer = 'Jessica'
city = 'Los Angeles'
```

- **Accessing Characters from Strings**: You can access characters from strings by using bracket notation like this:

```
my_str = "Hello world"

print(my_str[0])  # H
print(my_str[6])  # w

print(my_str[-1])  # d
print(my_str[-2]) # l
```

- **Escaping Strings**: You can use a backslash (`\`) if your string contains quotes like this:

```
msg = 'It\'s a sunny day'
quote = "She said, \"Hello!\""
```

- **String Concatenation**: To concatenate strings, you can use the `+` operator like this:

```
developer = 'Jessica'
print('My name is ' + developer + '.') # My name is Jessica
```

Another way to concatenate strings is by using the `+=` operator. This is used to perform concatenation and assignment in the same step like this:

```
greeting = 'My name is '
developer = 'Jessica.'

greeting += developer
print(greeting) # My name is Jessica.
```

- `f-strings`: This is short for formatted string literals. It allows you to handle interpolation and also do some concatenation with a compact and readable syntax:

```
developer = 'Jessica'
greeting = f'My name is {developer}.'
print(greeting) # My name is Jessica.
```

- **String Slicing**: This is when you can extract portions of a string. Here is the basic syntax:

```
str[start:stop:step]
```

The start position represents the index where the extraction should be begin. The stop position is where the slice should end. This position is non inclusive. The step position represents the interval to increment for the slicing. Here are some examples:

```

```

```
message = 'Python is fun!'

print(message[0:6])  # Python
print(message[7:])  # is fun!
print(message[::2])  # Pto sfn
```

- **Getting the Length of a String**: The `len()` function is used to return the number of the characters in the string:

```
developer = 'Jessica'

print(len(developer)) # 7
```

Working with the `in` operator

- **`in` Operator**: This returns a boolean that specifies whether the character or characters exist in the string or not:

```
my_str = 'Hello world'

print('Hello' in my_str)  # True
print('hey' in my_str)    # False
print('hi' in my_str)    # False
print('e' in my_str)  # True
print('f' in my_str)  # False
```

Common String Methods

- **`str.upper()`**: This returns a new string with all characters converted to uppercase:

```
developer = 'Jessica'

print(developer.upper()) # JESSICA
```

- **`str.lower()`**: This returns a new string with all characters converted to lowercase:

```
developer = 'Jessica'

print(developer.lower()) # jessica
```

- **`str.strip()`**: This returns a copy of the string with specified leading and trailing characters removed (if no argument is passed to the method, it removes leading and trailing whitespace).

```
greeting = '  hello world  '

trimmed_my_str = greeting.strip()
print(trimmed_my_str)  # 'hello world'
```

- **`replace()`**: This returns a new string with all occurrences of the old string replaced by a new one.

```

```

```
greeting = 'hello world'

replaced_my_str = greeting.replace('hello', 'hi')
print(replaced_my_str)  # 'hi world'
```

- `str.split()`: This is used to split a string into a list using a specified separator. A separator is a string specifying where the split should happen.

```
dashed_name = 'example-dashed-name'

split_words = dashed_name.split('-')
print(split_words)  # ['example', 'dashed', 'name']
```

- `str.join()`: This is used to join elements of an iterable into a string with a separator. An iterable is a collection of elements that can be looped over like a list, string or a tuple.

```
example_list = ['example', 'dashed', 'name']

joined_str = ' '.join(example_list)
print(joined_str)  # example dashed name
```

- `str.startswith(prefix)`: This returns a boolean indicating if a string starts with the specified prefix:

```
developer = 'Naomi'

result = developer.startswith('N')
print(result)  # True
```

- `str.endswith(suffix)`: This returns a boolean indicating if a string ends with the specified suffix:

```
developer = 'Naomi'

result = developer.endswith('N')
print(result)  # False
```

- `str.find()`: This returns the index for the first occurrence of a substring. If one is not found, then `-1` is returned:

```
developer = 'Naomi'

result = developer.find('N')
print(result)  # 0

city = 'Los Angeles'
print(city.find('New')) # -1
```

- `str.count(substring)`: This counts how many times a substring appears in a string:

```
city = 'Los Angeles'
print(city.count('e')) # 2
```

- **`str.capitalize()`**: This returns a new string with the first character capitalized and the other characters lowercased:

```
dessert = 'chocolate cake'
print(dessert.capitalize()) # Chocolate cake
```

- **`str.isupper()`**: This returns `True` if all letters in the string are uppercase and `False` if otherwise:

```
dessert = 'chocolate cake'
print(dessert.isupper()) # False
```

- **`str.islower()`**: This returns `True` if all letters in the string are lowercase and `False` if otherwise:

```
dessert = 'chocolate cake'
print(dessert.islower()) # True
```

- **`str.title()`**: This returns a new string with the first letter of each word capitalized:

```
city = 'los angeles'
print(city.title()) # Los Angeles
```

- **`str.maketrans()`**: This method is used to create a table of 1 to 1 character mappings for translation. It is often used with the `translate()` method which applies that table to a string and return the translated result.

```
trans_table = str.maketrans('abc', '123')
print(trans_table) # {97: 49, 98: 50, 99: 51}

result = 'abcabc'.translate(trans_table)
print(result)  # 123123
```

Common Operations used with Integers and Floats

- **Basic Math Operations**: In Python, you can do basic math operations with integers and floats including addition, subtraction, multiplication and division:

```
int_1 = 56
int_2 = 12
float_1 = 5.4
float_2 = 12.0

# Addition

print('Integer Addition:', int_1 + int_2) # Integer Addition: 68
print('Float Addition:', float_1 + float_2) # Float Addition: 17.4

# Subtraction

print('Int Subtraction:', int_1 - int_2) # Int Subtraction: 44
print('Float Subtraction:',  float_2 - float_1) # Float Subtraction: 6.6
```

```
# Multiplication

print('Int Multiplication:', int_1 * int_2) # Int Multiplication: 672
print('Float Multiplication:', float_2 * float_1) # Float Multiplication:
64.80000000000001

# Division

print('Int Division:', int_1 / int_2) # Int Division: 4.666666666666667
print('Float Division:', float_2 / float_1) # Float Division:
2.222222222222222
```

When you add a float and an integer, the result will be converted to a float like this:

```
int_1 = 56
float_1 = 5.4

print(int_1 + float_1) # 61.4
```

- **Modulus Operator (`%`)**: This returns the remainder when a number is divided by another number:

```
int_1 = 56
int_2 = 12

print(int_1 % int_2) # 8
```

- **Floor Division (`//`)**: This operator is used to divide two numbers and round down the result to the nearest whole number:

```
int_1 = 56
int_2 = 12

print(int_1 // int_2) # 4
```

- **Exponentiation Operator (`**`)**: This operator is used to raise a number to the power of another:

```
int_1 = 4
int_2 = 2

print(int_1 ** int_2) # 16
```

- **`float()` Function**: You can use this function to convert an integer to float.

```
num = 4

print(float(num)) # 4.0
```

- **`int()` Function**: You can use this function to convert an float to an integer.

```
num = 4.0

print(int(num)) # 4
```

- `round()` **Function**: This is used to round a number to the nearest whole integer:

```
num_1 = 3.4
num_2 = 7.7

print(round(num_1)) # 3
print(round(num_2)) # 8
```

- `abs()` **Function**: This is used to return the absolute value of a number:

```
num = -13

print(abs(num)) # 13
```

- `bin()` **Function**: This is used to convert an integer to its binary representation as a string:

```
num = 56

print(bin(num))  # 0b111000
```

- `oct()` **Function**: This is used to convert an integer to its octal representation as a string:

```
num = 56

print(oct(num))  # 0o70
```

- `hex()` **Function**: This is used to convert an integer to its hexadecimal representation as a string:

```
num = 56

print(hex(num))  # 0x38
```

- `pow()` **Function**: This is used to raise a number to the power of another:

```
result = pow(2, 3)
print(result)  # 8
```

Augmented Assignments

- **Definition**: Augmented assignment combines a binary operation with an assignment in one step. It takes a variable, applies an operation to it with another value, and stores the result back into the same variable.

```
# Addition assignment
my_var = 10
my_var += 5

print(my_var) # 15
```

```
# Subtraction assignment
count = 14
count -= 3

print(count) # 11

# Multiplication assignment
product = 65
product *= 7

print(product) # 455

# Division assignment
price = 100
price /= 4

print(price) # 25.0

# Floor Division assignment
total_pages = 23
total_pages //= 5

print(total_pages) # 4

# Modulus assignment
bits = 35
bits %= 2

print(bits) # 1

# Exponentiation assignment
power = 2
power **= 3

print(power) # 8
```

There are other augmented assignment operators too, like those for bitwise operators. They include `&=`, `^=`, `>>=`, and `<<=`.

Working with Functions

- **Definition**: Functions are reusable pieces of code that take inputs (arguments) and returns an output. To call a function, you need to reference the function name followed by a set of parenthesis:

```
# Defining a function

def get_sum(num_1, num_2):
    return num_1 + num_2

result = get_sum(3, 4) # function call
print(result) # 7
```

If a function does not explicitly return a value, then the default return value is `None`:

```
def greet():
    print('hello')

result = greet() # hello
print(result) # None
```

You can also supply default values to parameters like this:

```
def get_sum(num_1, num_2=2):
    return num_1 + num_2

result = get_sum(3)
print(result) # 5
```

If you call the function without the correct number of arguments, you will get a `TypeError`:

```
def calculate_sum(a, b):
    print(a + b)

calculate_sum()

# TypeError: calculate_sum() missing 2 required positional arguments: 'a' and
'b'
```

Common Built-in Functions

- `input()` **Function**: This is used to prompt the user for some input:

```
name = input('What is your name?') # User types 'Kolade' and presses Enter
print('Hello', name) # Hello Kolade
```

- `int()` **Function**: This is used to convert a number, boolean, or a numeric string into an integer:

```
print(int(3.14)) # 3
print(int('42')) # 42
print(int(True)) # 1
print(int(False)) # 0
```

Decorators

- **Definition**: Decorators are a special kind of function in Python. They are like wrappers for other functions, so they take another function as an argument.

```
def say_hello():
    name = input('What is your name? ')
    return 'Hello ' + name

def uppercase_decorator(func):
    def wrapper():
        original_func = func()
        modified_func = original_func.upper()
        return modified_func
    return wrapper

say_hello_res = uppercase_decorator(say_hello)

print(say_hello_res())
```

Scope in Python

- **Local Scope**: This is when a variable declared inside a function or class can only be accessed within that function or class.

```
def my_func():
    num = 10
    print(num)
```

- **Enclosing Scope**: This is when a function that's nested inside another function can access the variables of the function it's nested within.

```
def outer_func():
    msg = 'Hello there!'

    def inner_func():
        print(msg)
    inner_func()

print(outer_func()) # Hello there!
```

- **Global Scope**: This refers to variables that are declared outside any functions or classes which can be accessed from anywhere in the program.

```
tax = 0.70

def get_total(subtotal):
    total = subtotal + (subtotal * tax)
    return total

print(get_total(100))  # 170.0
```

- **Built-in Scope**: Reserved names in Python for predefined functions, modules, keywords, and objects.

```
print(str(45)) # '45'
print(type(3.14)) # <class 'float'>
print(isinstance(3, str)) # False
```

Comparison Operators

- **Equal (==)**: Checks if two values are equal:

```
print(3 == 4) # False
```

- **Not equal (!=)**: Checks if two values are not equal:

```
print(3 != 4) # True
```

- **Strictly greater than (>)**: Checks if one value is greater than another:

```
print(3 > 4) # False
```

- **Strictly less than (`<`)**: Checks if one value is less than another:

```
print(3 < 4) # True
```

- **Greater than or equal(`>=`)**: Checks if one value is greater than or equal to another:

```
print(3 >= 4) # False
```

- **Less than or equal(`<=`)**: Checks if one value is less than or equal to another:

```
print(3 <= 4) # True
```

Working with `if`, `elif` and `else` Statements

- **`if` Statements**: These are conditions used to determine if something is true or not. If the condition evaluates to `True`, then that block of code will run.

```
age = 18

if age >= 18:
    print('You are an adult') # You are an adult
```

- **`elif` Statement**: These are conditions that come after an `if` statement. If the `elif` condition evaluates to `True`, then that block of code will run.

```
age = 16

if age >= 18:
    print('You are an adult')
elif age >= 13:
    print('You are a teenager')  # You are a teenager
```

- **`else` Clause**: This will run if no other conditions evaluate to `True`.

```
age = 12

if age >= 18:
    print('You are an adult')
elif age >= 13:
    print('You are a teenager')
else:
    print('You are a child')  # You are a child
```

You can also use nested `if` statements like this:

```
is_citizen = True
age = 25

if is_citizen:
    if age >= 18:
        print('You are eligible to vote') # You are eligible to vote
```

```
else:
    print('You are not eligible to vote')
```

Truthy and Falsy Values

- **Definition**: In Python, every value has an inherent boolean value, or a built-in sense of whether it should be treated as `True` or `False` in a logical context. Many values are considered truthy, that is, they evaluate to `True` in a logical context. Others are falsy, meaning they evaluate to `False`. Here are some examples of falsy values:

```
None
False
Integer 0
Float 0.0
Empty strings ''
```

Other values like non-zero numbers, and non-empty strings are truthy.

Working with the `bool()` Function

- **Definition**: If you want to check whether a value is truthy or falsy, you can use the built-in `bool()` function. It explicitly converts a value to its boolean equivalent and returns `True` for truthy values and `False` for falsy values. Here are a few examples:

```
print(bool(False)) # False
print(bool(0))  # False
print(bool('')) # False

print(bool(True)) # True
print(bool(1)) # True
print(bool('Hello')) # True
```

Boolean Operators and Short-circuiting

- **Definition**: These are special operators that allow you to combine multiple expressions to create more complex decision-making logic in your code. There are three Boolean operators in Python: `and`, `or`, and `not`.
- **`and` Operator**: This operator takes two operands and returns the first operand if it is falsy, otherwise, it returns the second operand. Both operands must be truthy for an expression to result in a truthy value.

```
is_citizen = True
age = 25

print(is_citizen and age) # 25
```

You can also use the `and` operator in conditionals like this:

```
is_citizen = True
age = 25

if is_citizen and age >= 18:
    print('You are eligible to vote') # You are eligible to vote
```

```
else:
    print('You are not eligible to vote')
```

- **`or` Operator**: This operator returns the first operand if it is truthy, otherwise, it returns the second operand. An or expression results in a truthy value if at least one operand is truthy. Here is an example:

```
age = 19
is_employed = False

print(age or is_employed) # 19
```

Just like with the `and` operator, you can use the `or` operator in conditionals like this:

```
age = 19
is_student = True

if age < 18 or is_student:
    print('You are eligible for a student discount') # You are eligible for a
student discount
else:
    print('You are not eligible for a student discount')
```

- **Short-circuiting**: The `and` and `or` operators are known as a short-circuit operators. Short-circuiting means Python checks values from left to right and stops as soon as it determines the final result.
- **`not` Operator**: This operator takes a single operand and inverts its boolean value. It converts truthy values to `False` and falsy values to `True`. Unlike the previous operators we looked at, `not` always returns `True` or `False`. Here are some examples:

```
print(not '') # True, because empty string is falsy
print(not 'Hello') # False, because non-empty string is truthy
print(not 0) # True, because 0 is falsy
print(not 1) # False, because 1 is truthy
print(not False) # True, because False is falsy
print(not True) # False, because True is truthy
```

Here is an example of the `not` operator in a conditional:

```
is_admin = False

if not is_admin:
    print('Access denied for non-administrators.') # Access denied for non-
administrators.
else:
    print('Welcome, Administrator!')
```

--assignment--

Review the Python Basics topics and concepts.

React Basics Review

JavaScript Libraries and Frameworks

- JavaScript libraries and frameworks offer quick solutions to common problems and speed up development by providing pre-built code.
- Libraries are generally more focused on providing solutions to specific tasks, such as manipulating the DOM, handling events, or managing AJAX requests.
- A couple of examples of JavaScript libraries are jQuery and React.
- Frameworks, on the other hand, provide a more defined structure for building applications. They often come with a set of rules and conventions that developers need to follow.
- Examples of frameworks include Angular and Next.js, a meta framework for React.
- **Single page applications** (SPAs) are web applications that load a single HTML page and dynamically update that page as the user interacts with the application without reloading the entire page.
- SPAs use JavaScript to manage the application's state and render content. This is often done using frameworks which provide great tools for building complex user interfaces.
- Some issues surrounding SPAs include:
    - Screen readers struggle with dynamically updated content.
    - The URL does not change when the user navigates within the application, which can make it difficult to bookmark, backtrack or share specific pages.
    - Initial load time can be slow if the application is large as all the assets need to be loaded upfront.

React

- React is a popular JavaScript library for building user interfaces and web applications.
- A core concept of React is the creation of reusable UI components that can update and render independently as data changes.
- React allows developers to describe how the UI should look like based on the application state. React then updates and renders the right components when the data or the state changes.

React Components

- Components are the building blocks of React applications that allow developers to break down complex user interfaces into smaller, manageable pieces.
- The UI is described using JSX, an extension of JavaScript's syntax, that allows developers to write HTML-like code within JavaScript.
- Components are basically JS functions or classes that return a piece of UI.

Here is an example of a simple React component that renders a greeting message:

```
function Greeting() {
  const name = 'Anna';
  return <h1>Welcome, {name}!</h1>;
}
```

To use the component, you can simply call:

```
  <Greeting />
```

Importing and Exporting React components

- React components can be exported from one file and imported into another file.
- Let's say you have a component named `City` in a file named `City.js`. You can export the component using the `export` keyword:

```
// City.js
function City() {
  return <p>New York</p>;
}

export default City;
```

- To import the `City` component into another file, you can use the `import` keyword:

```
// App.js
import City from './City';

function App() {
  return (
    <div>
      <h1>My favorite city is:</h1>
      <City />
    </div>
  );
}
```

- The `default` keyword is used as it is the default export from the `City.js` file.
- You can also choose to export the component on the same line as the component definition like this:

```
export default function City() {
  return <p>New York</p>;
}
```

Setting up a React project using Vite

- Project setup tools and CLIs provide a quick & easy way to start new projects, allowing developers to focus on writing code rather than dealing with configuration.
- Vite, a popular project setup tool and can be used with React.
- To create a new project with Vite, you can use the following command in your terminal:

```
npm create vite@latest my-react-app -- --template react
```

This command creates a new React project named `my-react-app` using Vite's React template. In the project directory, you will see a `package.json` file with the project dependencies and commands listed in it.

- To run the project, navigate to the project directory and run the following commands:

```
cd my-react-app # path to the project directory
npm install # installs the dependencies listed in the package.json file
```

- Once the dependencies are installed, you should notice a new folder in your project called `node_modules`.

- The `node_modules` folder is where all the packages and libraries required by your project are stored.
- To run your project, use the following command:

```
npm run dev
```

- After that, open your browser and navigate to `http://localhost:5173` to see your React application running.
- To actually see the code for the starter template, you can go into your project inside the `src` folder and you should see the `App.jsx` file.

Passing props in React components

- In React, props (short for properties) are a way to pass data from a parent component to a child component. This mechanism is needed to create reusable and dynamic UI elements.
- Props can be any JavaScript value. To pass props from a parent to a child component, you add the props as attributes when you use the child component in the parent's JSX. Here's a simple example:

```
// Parent component
function Parent() {
  const name = 'Anna';
  return <Child name={name} />;
}

// Child component
function Child(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

You can pass multiple props using the spread operator `(...)`, after converting them to an object. Here's an example:

```
// Parent component
function Parent() {
  const person = {
    name: 'Anna',
    age: 25,
    city: 'New York'
  };
  return <Child {...person} />;
}
```

In this code, the spread operator `{...person}` converts the person object into individual props that are passed to the Child component.

Conditional rendering in React

- Conditional rendering in React allows you to create dynamic user interfaces. It is used to show different content based on certain conditions or states within your application.
- There are several ways to conditionally render content in React. One common approach is to use the ternary operator. Here's an example:

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
```

```
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please log in</h1>}
    </div>
  );
}
```

- Another way to conditionally render content is to use the logical AND (&&) operator. This is useful when you want to render content only if a certain condition is met. Here's an example:

```
function Greeting({ user }) {
  return (
    <div>
      {user && <h1>Welcome, {user.name}!</h1>}
    </div>
  );
}
```

In the code above, the `h1` element is only rendered if the user object is truthy.

You can also use a direct `if` statement this way:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please sign in</h1>;
}
```

Rendering lists in React

- Rendering lists in React is a common task when building user interfaces.
- Lists can be rendered using the JS array `map()` method to iterate over an array of items and return a new array of JSX elements.
- For example, if you have an array of names that you want to render as a list, you can do the following:

```
function NameList({ names }) {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={`${name}-${index}`}>{name}</li>
      ))}
    </ul>
  );
}
```

- Always remember to provide a unique key for each list item to help React manage the updating and rendering roles. With these techniques, you can create flexible, efficient, and dynamic lists in your React applications.

Inline styles in React

- Inline styles in React allow you to apply CSS styles directly to JSX elements using JavaScript objects.
- To apply inline styles in React, you can use the style attribute on JSX elements. The style attribute takes an object where the keys are CSS properties in camelCase and the values are the corresponding values. Here's an example:

```
function Greeting() {
  return (
    <h1
      style={{ color: 'blue', fontSize: '24px', backgroundColor: 'lightgray'
}}
    >
      Hello, world!
    </h1>
  );
}

export default Greeting;
```

You can also extract the styles into a separate object and reference it in the `style` attribute this way:

```
function Greeting() {
  const styles = {
    color: 'blue',
    fontSize: '24px',
    backgroundColor: 'lightgray'
  };

  return <h1 style={styles}>Hello, world!</h1>;
}

export default Greeting;
```

- Inline styles support dynamic styling by allowing you to conditionally apply styles based on props or state. Here is an example of how you can conditionally apply styles based on a prop:

```
function Greeting({ isImportant }) {
  const styles = {
    color: isImportant ? 'red' : 'black',
    fontSize: isImportant ? '24px' : '16px'
  };

  return <h1 style={styles}>Hello, world!</h1>;
}

export default Greeting;
```

- In the code above, the `color` and `fontSize` styles are conditionally set based on the `isImportant` prop.

--assignment--

Review the React Basics topics and concepts.

React Forms, Data Fetching and Routing Review

Working with Forms in React

- **Controlled Inputs**: This is when you store the input field value in state and update it through `onChange` events. This gives you complete control over the form data and allows instant validation and conditional rendering.

```
import { useState } from "react";

function App() {
  const [name, setName] = useState("");

  const handleChange = (e) => {
    setName(e.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(name);
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <label htmlFor="name">Your name</label> <br />
        <input value={name} id="name" onChange={handleChange} type="text" />
        <button type="submit">Submit</button>
      </form>
    </>
  );
}

export default App;
```

- **Uncontrolled Inputs**: Instead of handling the inputs through the `useState` hook, uncontrolled inputs in HTML maintain their own internal state with the help of the DOM. Since the DOM controls the input values, you need to pull in the values of the input fields with a `ref`.

```
import { useRef } from "react";

function App() {
 const nameRef = useRef();

 const handleSubmit = (e) => {
   e.preventDefault();
   console.log(nameRef.current.value);
 };

 return (
   <form onSubmit={handleSubmit}>
     <label htmlFor="name">Your</label>{" "}
     <input type="text" ref={nameRef} id="name" />
     <button type="submit">Submit</button>
   </form>
 );
}

export default App;
```

Working with the `useActionState` Hook

- **Server Actions**: These are functions that run on the server to allow form handling right on the server without the need for API endpoints. Here is an example from a Next.js application:

```
"use server";
```

```
async function submitForm(formData) {
 const name = formData.get("name");
 return { message: `Hello, ${name}!` };
}
```

The `"user server"` directive marks the function as a server action.

- **`useActionState` Hook**: This hook updates state based on the outcome of a form submission. Here's the basic syntax of the `useActionState` hook:

```
const [state, action, isPending] = useActionState(actionFunction,
initialState, permalink);
```

- `state` is the current state the action returns.
- `action` is the function that triggers the server action.
- `isPending` is a boolean that indicates whether the action is currently running or not.
- `actionFunction` parameter is the server action itself.
- `initialState` is the parameter that represents the starting point for the state before the action runs.
- `permalink` is an optional string that contains the unique page URL the form modifies.

Data Fetching in React

- **Options For Fetching Data**: There are many different ways to fetch data in React. You can use the native Fetch API, or third party tools like Axios or SWR.
- **Commonly Used State Variables When Fetching Data**: Regardless of which way you choose to fetch your data in React, there are some pieces of state you will need to keep track of. The first is the data itself. The second will track whether the data is still being fetched. The third is a state variable that will capture any errors that might occur during the data fetching process.

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
```

Since data fetching is a side effect, it's best to use the `Fetch API` inside of a `useEffect` hook.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/posts");

      if (!res.ok) {
        throw new Error("Network response was not ok");
      }

      const data = await res.json();
      setData(data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);
```

Then you can render a loading message if the data fetching is not complete, an error message if there was an error fetching the data, or the results.

```
if (loading) {
  return <p>Loading...</p>;
}

if (error) {
  return <p>{error.message}</p>;
}

return (
  <ul>
    {data.map((post) => (
      <li key={post.id}>{post.title}</li>
    ))}
  </ul>
);
```

If you want to use Axios, you need to install and import it:

```
npm i axios
```

```
import axios from "axios";
```

Then you can fetch the data using `axios.get`:

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await axios.get(
        "https://jsonplaceholder.typicode.com/users"
      );
      setData(res.data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);
```

To fetch data using the `useSWR` hook, you need to first install and import it.

```
npm i swr
```

```
import useSWR from "swr";
```

Here is how you can use the hook to fetch data:

```
import useSWR from "swr";

const fetcher = (url) => fetch(url).then((res) => res.json());

const FetchTodos = () => {
 const { data, error } = useSWR(
   "https://jsonplaceholder.typicode.com/todos",
   fetcher
 );

 if (!data) {
   return <h2>Loading...</h2>;
 }
 if (error) {
   return <h2>Error: {error.message}</h2>;
 }

 return (
   <>
     <h2>Todos</h2>
     <div>
       {data.map((todo) => (
         <h3 key={todo.id}>{todo.title}</h3>
       ))}
     </div>
   </>
 );
};

export default FetchTodos;
```

Working with the `useOptimistic` Hook

- **`useOptimistic` Hook**: This hook is used to keep UIs responsive while waiting for an async action to complete in the background. It helps manage "optimistic updates" in the UI, a strategy in which you provide immediate updates to the UI based on the expected outcome of an action, like waiting for a server response.

Here is the basic syntax:

```
const [optimisticState, addOptimistic] = useOptimistic(actualState,
updateFunction);
```

- `optimisticState` is the temporary state that updates right away for a better user experience.
- `addOptimistic` is the function that applies the optimistic update before the actual state changes.
- `actualState` is the real state value that comes from the result of an action, like fetching data from a server.
- `updateFunction` is the function that determines how the optimistic state should update when called.

Here is an example of using the `useOptimistic` hook in a `TaskList` component:

```
"use client";

import { useOptimistic } from "react";
```

```
export default function TaskList({ tasks, addTask }) {
  const [optimisticTasks, addOptimisticTask] = useOptimistic(
    tasks,
    (state, newTask) => [...state, { text: newTask, pending: true }]
  );

  async function handleSubmit(e) {
    e.preventDefault();
    const formData = new FormData(e.target);

    addOptimisticTask(formData.get("task"));

    addTask(formData);
    e.target.reset();
  }

  return <>{/* UI */}</>;
}
```

- `startTransition`: This is used to render part of the UI and mark a state update as a non-urgent transition. This allows the UI to be responsive during expensive updates. Here is the basic syntax:

```
startTransition(action);
```

The `action` performs a state update or triggers some transition-related logic. This ensures that urgent UI updates (like typing or clicking) are not blocked.

Working with the `useMemo` Hook

- **Memoization**: This is an optimization technique in which the result of expensive function calls are cached (remembered) based on specific arguments. When the same arguments are provided again, the cached result is returned instead of re-computing the function.
- `useMemo` **Hook**: This hook is used to memoize computed values. Here is an example of memoizing the result of sorting a large array. The `expensiveSortFunction` will only run when `largeArray` changes:

```
const memoizedSortedArray = useMemo(
  () => expensiveSortFunction(largeArray),
  [largeArray]
);
```

Working with the `useCallback` Hook

- `useCallback` **Hook**: This is used to memoize function references.

```
const handleClick = useCallback(() => {
  // code goes here
}, [dependency]);
```

- `React.memo`: This is used to memoize a component to prevent it from unnecessary re-renders when its prop has not changed.

```
const MemoizedComponent = React.memo(({ prop }) => {
 return (
```

```
    <>
      {/* Presentation */}
    </>
  )
});
```

Dependency Management Tools

- **Dependency Definition**: In software, a dependency is where one component or module in an application relies on another to function properly. Dependencies are common in software applications because they allow developers to use pre-built functions or tools created by others. The two core dependencies needed for a React project will be the `react` and `react-dom` packages:

```
"dependencies": {
  "react": "^18.3.1",
  "react-dom": "^18.3.1"
}
```

- **Package Manager Definition**: To manage software dependencies in a project, you will need to use a package manager. A package manager is a tool used for installation, updates and removal of dependencies. Many popular programming languages like JavaScript, Python, Ruby and Java, all use package managers. Popular package managers for JavaScript include npm, Yarn and pnpm.
- **`package.json` File**: This is a key configuration file in projects that contains metadata about your project, including its name, version, and dependencies. It also defines scripts, licensing information, and other settings that help manage the project and its dependencies.
- **`package-lock.json` File**: This file will lock down the exact versions of all packages that your project is using. When you update a package, then the new versions will be updated in the lock file as well.
- **`node_modules` Folder**: This folder contains the actual code for the dependencies listed in your `package.json` file, including both your project's direct dependencies and any dependencies of those dependencies.
- **Dev Dependencies**: These are packages that are only used for development and not in production. An example of this would be a testing library like Jest. You would install Jest as a dev dependency because it is needed for testing your application locally but not needed to have the application run in production.

```
"devDependencies": {
  "@eslint/js": "^9.17.0",
  "@types/react": "^18.3.18",
  "@types/react-dom": "^18.3.5",
  "@vitejs/plugin-react": "^4.3.4",
  "eslint": "^9.17.0",
  "eslint-plugin-react": "^7.37.2",
  "eslint-plugin-react-hooks": "^5.0.0",
  "eslint-plugin-react-refresh": "^0.4.16",
  "globals": "^15.14.0",
  "vite": "^6.0.5"
}
```

React Router

- **Introduction**: React Router is a third party library that allows you to add routing to your React applications. To begin, you will need to install React Router in an existing React project like this:

```
npm i react-router
```

Then inside of the `main.jsx` or `index.jsx` file, you will need to setup the route structure like this:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router";
import App from "./App.jsx";

import "./index.css";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<App />} />
      </Routes>
    </BrowserRouter>
  </StrictMode>
);
```

The `path` and `element` are used to couple the URL and UI components together. In this case, we are setting up a route for the homepage that points to the `App` component.

- **Multiple Views and Route Setup**: It is common in larger applications to have multiple views and routes setup like this:

```
<Routes>
  <Route index element={<Home />} />
  <Route path="about" element={<About />} />

  <Route path="products">
    <Route index element={<ProductsHome />} />
    <Route path=":category" element={<Category />} />
    <Route path=":category/:productId" element={<ProductDetail />} />
    <Route path="trending" element={<Trending />} />
  </Route>
</Routes>
```

The `index` prop in these examples is meant to represent the default route for a given path segment. So the `Home` component will be shown at the root `/` path while the `ProductsHome` component will be shown at the `/products` path.

- **Nesting Routes**: You can nest routes inside other routes which results in the path of the child route being appended to the parent route's path.

```
<Route path="products">
  <Route path="trending" element={<Trending />} />
</Route>
```

In the example above, the path for the trending products will be products/trending.

- **Dynamic Segments**: A dynamic segment is where any part of the URL path is dynamic.

```
<Route path=":category" element={<Category />} />
```

In this example we have a dynamic segment called `category`. When a user navigates to a URL like products/brass-instruments, then the view will change to the `Category` component and you can

dynamically fetch the appropriate data based on the segment.

- `useParams` **Hook**: This hook is used to access the dynamic parameters from a URL path.

```
import { useParams } from "react-router";

export default function Category() {
  let params = useParams();
  {/* Accessing the category param: params.category */}
  {/* rest of code goes here */}
}
```

React Frameworks

- **Introduction**: React frameworks provide features like routing, image optimizations, data fetching, authentication and more. This means that you might not need to set up separate frontend and backend applications for certain use cases. Examples of React Frameworks include Next.js and Remix.
- **Next.js Routing**: This routing system includes support for dynamic routes, parallel routes, route handlers, redirects, internalization and more.

Here is an example of creating a custom request handler:

```
export async function GET() {
  const res = await fetch("https://example-api.com");
  const data = await res.json();

  return Response.json({ data });
}
```

- **Next.js Image Optimization**: The `Image` component extends the native HTML `img` element and allows for faster page loads and size optimizations. This means that images will only load when they enter the viewport and the `Image` component will automatically serve correctly sized images for each device.

```
import Image from "next/image";

export default function Page() {
  return (
    <Image src="link-to-image-goes-here" alt="descriptive-title-goes-here" />
  );
}
```

Prop Drilling

- **Definition**: Prop drilling is the process of passing props from a parent component to deeply nested child components, even when some of the child components don't need the props.

State Management

- **Context API**: Context refers to when a parent component makes information available to child components without needing to pass it explicitly through props. `createContext` is used to create a context object which represent the context that other components will read. The `Provider` is used to supply context values to the child components.

```
```

```
import { useState, createContext } from "react";

const CounterContext = createContext();

const CounterProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  return (
    <CounterContext.Provider value={{ count, setCount }}>
      {children}
    </CounterContext.Provider>
  );
};

export { CounterContext, CounterProvider };
```

- **Redux**: Redux handles state management by providing a central store and strict control over state updates. It uses a predictable pattern with actions, reducers, and middleware. Actions are payloads of information that send data from your application to the Redux store, often triggered by user interactions. Reducers are functions that specify how the state should change in response to those actions, ensuring the state is updated in an immutable way. Middleware, on the other hand, acts as a bridge between the action dispatching and the reducer, allowing you to extend Redux's functionality (e.g., logging, handling async operations) without modifying the core flow.

- **Zustand**: This state management solution is ideal for small to medium-scale applications. It works by using a `useStore` hook to access access state directly in components and pages. This lets you modify and access data without needing actions, reducers, or a provider.

Debugging React Components Using the React DevTools

- **React Developer Tools**: This is a browser extension you can use in Chrome, Firefox and Edge to inspect React components and identify performance issues. For Safari, you will need to install the `react-devtools` npm package. After installing React DevTools and opening a React app in the browser, open the browser developer tools to access the two extra tabs provided for debugging React – Components and Profiler.
- **Components Tab**: This tab displays each component for you in a tree view format. Here are some things you can do in this tab:
  - view the app's component hierarchy
  - check and modify props, states, and context values in real time
  - check the source code for each selected component
  - log the component data to the console
  - inspect the DOM elements for the component
- **Profiler Tab**: This tab helps you analyze component performance. You can record component performance so you can identify unnecessary re-renders, view commit durations, and subsequently optimize slow components.

React Server Components

- **Definition**: React Server Components are React components that render exclusively on the server, sending only the final HTML to the client. This means those components can directly access server-side resources and dramatically reduce the amount of JavaScript sent to the browser.

--assignment--

Review React routing, state management, forms and data fetching.

React State and Hooks Review

Working with Events in React

- **Synthetic Event System**: This is React's way of handling events. It serves as a wrapper around the native events like the `click`, `keydown`, and `submit` events. Event handlers in React use the camel casing naming convention. (Ex. `onClick`, `onSubmit`, etc)

Here is an example of using the `onClick` attribute for a `button` element in React:

```
function handleClick() {
  console.log("Button clicked!");
}

<button onClick={handleClick}>Click Me</button>;
```

In React, event handler functions usually start with the prefix `handle` to indicate they are responsible for handling events, like `handleClick` or `handleSubmit`.

When a user action triggers an event, React passes a Synthetic Event object to your handler. This object behaves much like the native Event object in vanilla JavaScript, providing properties like `type`, `target`, and `currentTarget`.

To prevent default behaviors like browser refresh during an `onSubmit` event, for example, you can call the `preventDefault()` method:

```
function handleSubmit(event) {
  event.preventDefault();
  console.log("Form submitted!");
}

<form onSubmit={handleSubmit}>
  <input type="text" />
  <button>Submit</button>
</form>;
```

You can also wrap a handler function in an arrow function like this:

```
function handleDelete(id) {
  console.log("Deleting item:", id);
}

<button onClick={() => handleDelete(1)}>Delete Item</button>;
```

Working with State and the `useState` Hook

- **Definition for state**: In React, state is a object that contains data for a component. When the state updates the component will re-render. React treats state as immutable, meaning you should not modify it directly.
- **`useState()` Hook**: The `useState` hook is a function that lets you declare state variables in functional components. Here is an basic syntax:

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

In the state variable you have the following:

- `stateVariable` holds the current state value
- `setStateFunction` (the setter function) updates the state variable
- `initialValue` sets the initial state

Here is a complete example for a `Counter` component:

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>{count}</h2>

      <button onClick={() => setCount(count - 1)}>Decrement</button>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

Rendering and React Components

- **Definition**: In React, rendering is the process by which components appear in the user interface (UI), usually the browser. The rendering process consists of three stages: trigger, render, and commit.

The trigger stage occurs when React detects that something has changed and the user interface (UI) might need to be updated. This change is often due to an update in state or props.

Once the trigger happens, React enters the render stage. Here, React re-evaluates your components and figures out what to display. To do this, React uses a lightweight copy of the "real" DOM called the virtual DOM. With the virtual DOM, React can quickly check what needs to change in the component.

The commit stage is where React takes the prepared changes from the virtual DOM and applies them to the real DOM. In other words, this is the stage where you see the final result on the screen.

Updating Objects and Arrays in State

- **Updating Objects in State**: If you need to update an object in state, then you should make a new object or copy an existing object first, then set the state for that new object. Any object put into state should be considered as read-only. Here is an example of setting a user's name, age and city. The `handleChange` function is used to handle updates to the user's information:

```
import { useState } from "react";

function Profile() {
  const [user, setUser] = useState({ name: "John Doe", age: 31, city: "LA"
});

  const handleChange = (e) => {
    const { name, value } = e.target;

    setUser((prevUser) => ({...prevUser, [name]: value}));
  };

  return (
```

```
    <div>
      <h1>User Profile</h1>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
      <p>City: {user.city}</p>

      <h2>Update User Age </h2>
      <input type="number" name="age" value={user.age} onChange=
{handleChange} />

      <h2>Update User Name </h2>
      <input type="text" name="name" value={user.name} onChange=
{handleChange} />

      <h2>Update User City </h2>
      <input type="text" name="city" value={user.city} onChange=
{handleChange} />
    </div>
  );
}

export default Profile;
```

- **Updating Arrays in State**: When updating arrays in state, it is important not to directly modify the array using methods like `push()` or `pop()`. Instead you should create a new array when updating state:

```
const addItem = () => {
  const newItem = {
    id: items.length + 1,
    name: `Item ${items.length + 1}`,
  };

  // Creates a new array
  setItems((prevItems) => [...prevItems, newItem]);
};
```

If you want to remove items from an array, you should use the `filter()` method, which returns a new array after filtering out whatever you want to remove:

```
const removeItem = (id) => {
  setItems((prevItems) => prevItems.filter((item) => item.id !== id));
};
```

Referencing Values Using Refs

- **`ref` Attribute**: You can access a DOM node in React by using the `ref` attribute. Here is an example to showcase a `ref` to focus an `input` element. The `current` property is used to access the current value of that `ref`:

```
import { useRef } from "react";

const Focus = () => {
  const inputRef = useRef(null);

  const handleFocus = () => {
    if (inputRef.current) {
      inputRef.current.focus();
```

```
    }
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Enter text" />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );
};

export default Focus;
```

Working with the `useEffect` Hook

- **`useEffect()` Hook**: In React, an effect is anything that happens outside the component rendering process. That is, anything React does not handle directly as part of rendering the UI. Common examples include fetching data, updating the browser tab's title, reading from or writing to the browser's local storage, getting the user's location, and much more. These operations interact with the outside world and are known as side effects. React provides the `useEffect` hook to let you handle those side effects. `useEffect` lets you run a function after the component renders or updates.

```
import { useEffect } from "react";

useEffect(() => {
 // Your side effect logic (usually a function) goes here
}, [dependencies]);
```

The effect function runs after the component renders, while the optional `dependencies` argument controls when the effect runs.

Note that `dependencies` can be an array of "reactive values" (state, props, functions, variables, and so on), an empty array, or omitted entirely. Here's how all of those options control how `useEffect` works:

- If `dependencies` is an array that includes one or more reactive values, the effect will run whenever they change.
- If `dependencies` is an empty array, `useEffect` runs only once when the component first renders.
- If you omit `dependencies`, the effect runs every time the component renders or updates.

How to Create Custom Hooks

- **Custom Hooks**: A custom hook allows you to extract reusable logic from components, such as data fetching, state management, toggling, and side effects like tracking online status. In React, all built-in hooks start with the word `use`, so your custom hook should follow the same convention.

Here is an example of creating a `useDebounce` hook:

```
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
```

```
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

export { useDebounce };
```

--assignment--

Review the React state and hooks topics and concepts.

Recursion Review

- Recursion is a programming concept that allows you to call a function repeatedly until a base-case is reached.

Here is an example of a recursive function that calculates the factorial of a number:

```
function findFactorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * findFactorial(n - 1);
}
```

In the above example, the `findFactorial` function is called recursively until `n` reaches `0`. When `n` is `0`, the base case is reached and the function returns `1`. The function then returns the product of `n` and the result of the recursive call to `findFactorial(n - 1)`.

- Recursion allows you to handle something with an unknown depth, such as deeply nested objects/arrays, or a file tree.
- A call stack is used to keep track of the function calls in a recursive function. Each time a function is called, it is added to the call stack. When the base case is reached, the function calls are removed from the stack.
- You should carefully define the base case as calling it indefinitely can cause your code to crash. This is because the recursion keeps piling more and more function calls till the system runs out of memory.
- Recursions find their uses in solving mathematical problems like factorial and Fibonacci, traversing trees and graphs, generating permutations and combinations and much more.

--assignment--

Review the Recursion topics and concepts.

Relational Database Review

Introduction to Relational Databases

- **Relational Databases**: Organize data into related tables made of rows and columns. Each row represents a record, and each column represents an attribute of the data.

- **Advantages of Relational Databases**: Scalable, widely applicable across domains (e.g., healthcare, business, gaming), and structured to maintain reliable data.
- **Common Use Cases**: Web development, inventory systems, e-commerce, healthcare, and enterprise applications.

Key Concepts

- **Schema**: A relational database requires a schema that defines its structure—tables, columns, data types, constraints, and relationships.
- **Primary Keys**: Unique identifiers for each row in a table. They are essential for data integrity and are used to relate records between tables via foreign keys.
- **Foreign Keys**: References to primary keys in another table, used to link related data across tables.
- **Relationships**: By connecting tables through primary and foreign keys, you can structure normalized data and perform meaningful queries.
- **Entity Relationship Diagrams (ERDs)**: Visualize how entities (tables) relate to each other in a database schema.
- **Data Integrity**: Enforced using keys and data types. Ensures consistency and accuracy of stored data.

SQL Basics

- **Queries**: Requests to retrieve specific data from the database.

```
SELECT * FROM dogs WHERE age < 3;
```

- **WHERE clause**: Filter results based on conditions. Use comparison operators like `<`, `=`, `>`, etc.
- **Select with ORDER BY**: Retrieve and sort results based on a column.

```
SELECT columns FROM table_name ORDER BY column_name;
```

Table Operations

- `CREATE TABLE` **Statement**: This statement is used to create a new table in a database.

```
CREATE TABLE first_table();
```

- `ALTER TABLE ADD COLUMN` **Statement**: This statement is used to add a column to an existing table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE;
```

- `ALTER TABLE DROP COLUMN` **Statement**: This statement is used to drop a column from an existing table.

```
ALTER TABLE table_name DROP COLUMN column_name;
```

- `ALTER TABLE RENAME COLUMN` **Statement**: This statement is used to rename a column in a table.

```
ALTER TABLE table_name RENAME COLUMN column_name TO new_name;
```

- `DROP TABLE` **Statement**: This statement is used to drop an entire table from the database.

```
DROP TABLE table_name;
```

- `ALTER DATABASE RENAME` **Statement**: This statement is used to rename a database.

```
ALTER DATABASE database_name RENAME TO new_database_name;
```

- `DROP DATABASE` **Statement**: This statement is used to drop an entire database.

```
DROP DATABASE database_name;
```

Constraints & Data Integrity

- `ALTER TABLE ADD COLUMN` **with Constraint**: This statement is used to add a column with a constraint to an existing table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE CONSTRAINT;
```

- `NOT NULL` **Constraint**: This constraint ensures that a column cannot have NULL values.

```
column_name VARCHAR(50) NOT NULL
```

- `ALTER TABLE ADD PRIMARY KEY` **Statement**: This statement is used to add a primary key constraint to a table.

```
ALTER TABLE table_name ADD PRIMARY KEY(column_name);
```

- `ALTER TABLE DROP CONSTRAINT` **Statement**: This statement is used to drop a constraint from a table.

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

- `ALTER TABLE ADD COLUMN` **with Foreign Key**: This statement is used to add a foreign key column that references another table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE REFERENCES
referenced_table_name(referenced_column_name);
```

- `ALTER TABLE ADD UNIQUE` **Statement**: This statement is used to add a UNIQUE constraint to a column.

```
ALTER TABLE table_name ADD UNIQUE(column_name);
```

- `ALTER TABLE ALTER COLUMN SET NOT NULL` **Statement**: This statement is used to set a NOT NULL constraint on an existing column.

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

- `INSERT` **Statement with NULL Values**: This statement demonstrates how to insert NULL values into a table.

```
INSERT INTO table_name(column_a) VALUES(NULL);
-- or
INSERT INTO table_name(column_b) VALUES('value'); -- if column_a allows nulls
```

- **Composite Primary Key**: This constraint defines a primary key that consists of multiple columns.

```
CREATE TABLE course_enrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Data Manipulation (CRUD)

- `INSERT` **Statement**: This statement is used to insert a single row into a table.

```
INSERT INTO table_name(column_1, column_2) VALUES(value1, value2);
```

- `INSERT` **Statement with Omitted Columns**: This statement shows how to insert values without explicitly listing the column names, relying on the default column order in the table.

```
INSERT INTO dogs VALUES ('Gino', 3);
```

- `INSERT` **Statement with Multiple Rows**: This statement is used to insert multiple rows into a table in a single operation.

```
INSERT INTO dogs (name, age) VALUES
('Gino', 3),
('Nora', 2);
```

- `UPDATE` **Statement**: This statement is used to update existing data in a table based on a condition.

```
UPDATE table_name SET column_name=new_value WHERE condition;
```

- `DELETE` **Statement**: This statement is used to delete rows from a table based on a condition.

```
DELETE FROM table_name WHERE condition;
```

Data Types

- `NUMERIC` **Data Type**: This data type is used to store precise decimal numbers with a specified precision and scale.

```
price NUMERIC(10, 2)
```

- `TEXT` **Data Type**: This data type is used to store variable-length character strings with no specific length limit.

```
column_name TEXT
```

- `INTEGER` **Data Type**: This data type is used to store whole numbers without decimal places.

```
units_sold INTEGER
```

- `SMALLINT` **and** `BIGINT` **Data Types**: These are variants of INTEGER with smaller and larger ranges respectively.
- `SERIAL` **Data Type**: This data type is used to create auto-incrementing integer columns in PostgreSQL.

```
id SERIAL
```

- `AUTO_INCREMENT` **Attribute**: This attribute is used in MySQL to create auto-incrementing integer columns.

```
id INT AUTO_INCREMENT
```

- `VARCHAR` **Data Type**: This data type is used to store variable-length character strings with a specified maximum length.

```
name VARCHAR(50)
```

- `DATE` **Data Type**: This data type is used to store date values (year, month, day).

```
event_date DATE
```

- `TIME` **Data Type**: This data type is used to store time values (hour, minute, second).

```
start_time TIME
```

- `TIMESTAMP` **Data Type**: This data type is used to store date and time values, optionally with time zone information.

```
event_timestamp TIMESTAMP
event_timestamp TIMESTAMP WITH TIME ZONE
```

- `BOOLEAN` **Data Type**: This data type is used to store true/false values.

```
is_active BOOLEAN
```

Database Relationships

- **Types of Relationships**: These are the different ways tables can be related to each other in a relational database.
    - One-to-one
    - One-to-many
    - Many-to-one
    - Many-to-many
    - Self-referencing (recursive)
- **One-to-One Relationship**: This relationship type means that each record in one table corresponds to exactly one record in another table.

```
One employee is assigned exactly one vehicle.
Tables: employees, vehicles
```

- **One-to-Many Relationship**: This relationship type means that one record in a table can be associated with multiple records in another table.

```
One customer can have many orders.
Tables: customers → orders
```

- **Many-to-Many Relationship via Junction Table**: This relationship type is implemented using a junction table that contains foreign keys from both related tables.

```
CREATE TABLE books_authors (
    author_id INT REFERENCES authors(id),
    book_id INT REFERENCES books(id)
);
```

- **Self-Referencing Relationships**: This relationship type occurs when a table references itself, creating a hierarchical structure.

```
An employee table where each employee may report to another employee.
```

Advanced SQL (Joins)

- `INNER JOIN` **Statement**: This join returns only the rows that have matching values in both tables.

```
SELECT *
FROM products
INNER JOIN sales ON products.product_id = sales.product_id;
```

- `FULL OUTER JOIN` **Statement**: This join returns all rows from both tables, including unmatched rows from either table.

```
SELECT *
FROM products
FULL OUTER JOIN sales ON products.product_id = sales.product_id;
```

- `LEFT OUTER JOIN` **Statement**: This join returns all rows from the left table and matching rows from the right table.

```
SELECT *
FROM products
LEFT JOIN sales ON products.product_id = sales.product_id;
```

- `RIGHT OUTER JOIN` **Statement**: This join returns all rows from the right table and matching rows from the left table.

```
SELECT *
FROM products
RIGHT JOIN sales ON products.product_id = sales.product_id;
```

- `SELF JOIN` **Statement**: This join is used to join a table with itself to compare rows within the same table.

```
SELECT A.column_name, B.column_name
FROM table_name A
JOIN table_name B ON A.related_column = B.related_column;
```

- `CROSS JOIN` **Statement**: This join returns the Cartesian product of two tables, combining every row from the first table with every row from the second table.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

PostgreSQL Specific Commands

- `psql` **Login Command**: This command is used to log in to PostgreSQL with specific username and database.

```
psql --username=freecodecamp --dbname=postgres
```

- `\l` **Command**: This command lists all databases in the PostgreSQL instance.

```
\l
```

- `CREATE DATABASE` **and** `\c` **Commands**: These commands are used to create a new database and connect to it.

```
CREATE DATABASE database_name;
\c database_name
```

- `\d` **Command**: This command lists all tables in the current database.

```
\d
```

- `\d table_name` **Command**: This command displays the schema/structure of a specific table.

```
\d table_name
```

- `\q` **Command**: This command exits the PostgreSQL client.

```
\q
```

Relational vs Non-Relational

- **Non-Relational (NoSQL) Databases**: Store unstructured or semi-structured data. Do not require a rigid schema and are more flexible for evolving data models.
- **Choosing Between Relational and Non-Relational**: Depends on the nature of your data and application requirements.
- **Relational vs Non-Relational**: Choose relational for structured data and consistency; NoSQL for flexibility and fast-changing data.

Popular RDBMS Systems

- **MySQL**: Open-source, reliable, widely used in web development, supported by a large community.
- **PostgreSQL**: Open-source, advanced, extensible. Supports custom data types and server-side programming.
- **SQLite**: Lightweight, file-based, serverless. Ideal for small applications.
- **Microsoft SQL Server**: Proprietary, enterprise-grade database.
- **Oracle Database**: Commercial RDBMS known for large-scale performance and scalability.

Best Practices

- **Naming Convention**: Use `snake_case` (e.g., `delivery_orders`) for table and column names.

--assignment--

Review the Relational Database topics and concepts.

Relational Databases Review

Terminal, Shell, and Command Line Basics

- **Command line**: A text interface where users type commands.
- **Terminal**: The application that provides access to the command line.
- **Terminal emulator**: Adds extra features to a terminal.
- **Shell**: Interprets the commands entered into the terminal (e.g., Bash).
- **PowerShell / Command Prompt / Microsoft Terminal**: Options for accessing the command line on Windows.
- **Terminal (macOS)**: Built-in option on macOS, with third-party alternatives like iTerm or Ghostty.
- **Terminal (Linux)**: Options vary by distribution, with many third-party emulators like kitty.

- **Terminology**: Though "terminal," "shell," and "command line" are often used interchangeably, they have specific meanings.

Command Line Shortcuts

- **Up/Down arrows**: Cycle through previous/next commands in history.
- **Tab**: Autocomplete commands.
- `Control+L` (Linux/macOS) or typing `cls` (Windows): Clear the terminal screen.
- `Control+C`: Interrupt a running command (also used for copying in PowerShell if text is selected).
- `Control+Z` (Linux/macOS only): Suspend a task to the background; use `fg` to resume it.
- `!!`: Instantly rerun the last executed command.

Bash Basics

- **Bash** (Bourne Again Shell): Widely used Unix-like shell.
  Key commands:
  - `pwd`: Show the current directory.
  - `cd`: Change directories.
    - `..` refers to the parent directory (one level up).
    - `.` refers to the current directory.
  - `ls`: List files and folders.
    - `-a`: Show all files, including hidden files.
    - `-l`: Show detailed information about files.
  - `less`: View file contents one page at a time with navigation options, including scrolling backward and searching.
  - `more`: Display file contents one screen at a time, with limited backward scrolling and basic navigation.
  - `cat`: Show the entire file content at once without scrolling or navigation, useful for smaller files.
  - `mkdir`: Create a new directory.
  - `rmdir`: Remove an empty directory.
  - `touch`: Create a new file.
  - `mv`: Move or rename files.
    - Rename: `mv oldname.txt newname.txt`
    - Move: `mv filename.txt /path/to/target/`
  - `cp`: Copy files.
    - `-r`: Recursively copy directories and their contents.
  - `rm`: Delete files.
    - `-r`: Recursively delete directories and their contents.
  - `echo`: Display a line of text or a variable's value.
    - Use `>` to overwrite the existing content in a file. (e.g., `echo "text" > file.txt`)
    - Use `>>` to append output to a file **without overwriting existing content** (e.g., `echo "text" >> file.txt`).
  - `exit`: Exit the terminal session.
  - `clear`: Clear the terminal screen.
  - `find`: Search for files and directories.
    - `-name`: Search for files by name pattern (e.g., `find . -name "*.txt"`).
  - Use `man` followed by a command (e.g., `man ls`) to access detailed manual/help pages.

Command Options and Flags

- **Options** or **flags**: modify a command's behavior and are usually prefixed with hyphens:
  - **Long form (two hyphens)**:
    - Example: `--help`, `--version`
    - Values are attached using an equals sign, e.g., `--width=50`.
  - **Short form (one hyphen)**:
    - Example: `-a`, `-l`

    - Values are passed with a space, e.g., `-w 50`.
    - Multiple short options can be chained together, e.g., `ls -alh`.
- `--help`: You can always use a command with this flag to understand the available options for any command.

Introduction to Relational Databases

- **Relational Databases**: Organize data into related tables made of rows and columns. Each row represents a record, and each column represents an attribute of the data.
- **Advantages of Relational Databases**: Scalable, widely applicable across domains (e.g., healthcare, business, gaming), and structured to maintain reliable data.
- **Common Use Cases**: Web development, inventory systems, e-commerce, healthcare, and enterprise applications.

Key Concepts

- **Schema**: A relational database requires a schema that defines its structure—tables, columns, data types, constraints, and relationships.
- **Primary Keys**: Unique identifiers for each row in a table. They are essential for data integrity and are used to relate records between tables via foreign keys.
- **Foreign Keys**: References to primary keys in another table, used to link related data across tables.
- **Relationships**: By connecting tables through primary and foreign keys, you can structure normalized data and perform meaningful queries.
- **Entity Relationship Diagrams (ERDs)**: Visualize how entities (tables) relate to each other in a database schema.
- **Data Integrity**: Enforced using keys and data types. Ensures consistency and accuracy of stored data.

SQL Basics

- **Queries**: Requests to retrieve specific data from the database.

```
SELECT * FROM dogs WHERE age < 3;
```

- **WHERE clause**: Filter results based on conditions. Use comparison operators like `<`, `=`, `>`, etc.
- **Select with ORDER BY**: Retrieve and sort results based on a column.

```
SELECT columns FROM table_name ORDER BY column_name;
```

Table Operations

- `CREATE TABLE` **Statement**: This statement is used to create a new table in a database.

```
CREATE TABLE first_table();
```

- `ALTER TABLE ADD COLUMN` **Statement**: This statement is used to add a column to an existing table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE;
```

- `ALTER TABLE DROP COLUMN` **Statement**: This statement is used to drop a column from an existing table.

```
ALTER TABLE table_name DROP COLUMN column_name;
```

- `ALTER TABLE RENAME COLUMN` **Statement**: This statement is used to rename a column in a table.

```
ALTER TABLE table_name RENAME COLUMN column_name TO new_name;
```

- `DROP TABLE` **Statement**: This statement is used to drop an entire table from the database.

```
DROP TABLE table_name;
```

- `ALTER DATABASE RENAME` **Statement**: This statement is used to rename a database.

```
ALTER DATABASE database_name RENAME TO new_database_name;
```

- `DROP DATABASE` **Statement**: This statement is used to drop an entire database.

```
DROP DATABASE database_name;
```

Constraints & Data Integrity

- `ALTER TABLE ADD COLUMN` **with Constraint**: This statement is used to add a column with a constraint to an existing table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE CONSTRAINT;
```

- `NOT NULL` **Constraint**: This constraint ensures that a column cannot have NULL values.

```
column_name VARCHAR(50) NOT NULL
```

- `ALTER TABLE ADD PRIMARY KEY` **Statement**: This statement is used to add a primary key constraint to a table.

```
ALTER TABLE table_name ADD PRIMARY KEY(column_name);
```

- `ALTER TABLE DROP CONSTRAINT` **Statement**: This statement is used to drop a constraint from a table.

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

- `ALTER TABLE ADD COLUMN` **with Foreign Key**: This statement is used to add a foreign key column that references another table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE REFERENCES
referenced_table_name(referenced_column_name);
```

- `ALTER TABLE ADD UNIQUE` **Statement**: This statement is used to add a UNIQUE constraint to a column.

```
ALTER TABLE table_name ADD UNIQUE(column_name);
```

- `ALTER TABLE ALTER COLUMN SET NOT NULL` **Statement**: This statement is used to set a NOT NULL constraint on an existing column.

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

- `INSERT` **Statement with NULL Values**: This statement demonstrates how to insert NULL values into a table.

```
INSERT INTO table_name(column_a) VALUES(NULL);
-- or
INSERT INTO table_name(column_b) VALUES('value'); -- if column_a allows nulls
```

- **Composite Primary Key**: This constraint defines a primary key that consists of multiple columns.

```
CREATE TABLE course_enrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Data Manipulation (CRUD)

- `INSERT` **Statement**: This statement is used to insert a single row into a table.

```
INSERT INTO table_name(column_1, column_2) VALUES(value1, value2);
```

- `INSERT` **Statement with Omitted Columns**: This statement shows how to insert values without explicitly listing the column names, relying on the default column order in the table.

```
INSERT INTO dogs VALUES ('Gino', 3);
```

- `INSERT` **Statement with Multiple Rows**: This statement is used to insert multiple rows into a table in a single operation.

```
INSERT INTO dogs (name, age) VALUES
('Gino', 3),
('Nora', 2);
```

- `UPDATE` **Statement**: This statement is used to update existing data in a table based on a condition.

```
UPDATE table_name SET column_name=new_value WHERE condition;
```

- `DELETE` **Statement**: This statement is used to delete rows from a table based on a condition.

```
DELETE FROM table_name WHERE condition;
```

Data Types

- `NUMERIC` **Data Type**: This data type is used to store precise decimal numbers with a specified precision and scale.

```
price NUMERIC(10, 2)
```

- `TEXT` **Data Type**: This data type is used to store variable-length character strings with no specific length limit.

```
column_name TEXT
```

- `INTEGER` **Data Type**: This data type is used to store whole numbers without decimal places.

```
units_sold INTEGER
```

- `SMALLINT` **and** `BIGINT` **Data Types**: These are variants of INTEGER with smaller and larger ranges respectively.
- `SERIAL` **Data Type**: This data type is used to create auto-incrementing integer columns in PostgreSQL.

```
id SERIAL
```

- `AUTO_INCREMENT` **Attribute**: This attribute is used in MySQL to create auto-incrementing integer columns.

```
id INT AUTO_INCREMENT
```

- `VARCHAR` **Data Type**: This data type is used to store variable-length character strings with a specified maximum length.

```
name VARCHAR(50)
```

- `DATE` **Data Type**: This data type is used to store date values (year, month, day).

```
event_date DATE
```

- `TIME` **Data Type**: This data type is used to store time values (hour, minute, second).

```
start_time TIME
```

- `TIMESTAMP` **Data Type**: This data type is used to store date and time values, optionally with time zone information.

```
event_timestamp TIMESTAMP
event_timestamp TIMESTAMP WITH TIME ZONE
```

- `BOOLEAN` **Data Type**: This data type is used to store true/false values.

```
is_active BOOLEAN
```

Database Relationships

- **Types of Relationships**: These are the different ways tables can be related to each other in a relational database.
    - One-to-one
    - One-to-many
    - Many-to-one
    - Many-to-many
    - Self-referencing (recursive)
- **One-to-One Relationship**: This relationship type means that each record in one table corresponds to exactly one record in another table.

```
One employee is assigned exactly one vehicle.
Tables: employees, vehicles
```

- **One-to-Many Relationship**: This relationship type means that one record in a table can be associated with multiple records in another table.

```
One customer can have many orders.
Tables: customers → orders
```

- **Many-to-Many Relationship via Junction Table**: This relationship type is implemented using a junction table that contains foreign keys from both related tables.

```
CREATE TABLE books_authors (
    author_id INT REFERENCES authors(id),
    book_id INT REFERENCES books(id)
);
```

- **Self-Referencing Relationships**: This relationship type occurs when a table references itself, creating a hierarchical structure.

```
An employee table where each employee may report to another employee.
```

Advanced SQL (Joins)

- `INNER JOIN` **Statement**: This join returns only the rows that have matching values in both tables.

```
SELECT *
FROM products
INNER JOIN sales ON products.product_id = sales.product_id;
```

- `FULL OUTER JOIN` **Statement**: This join returns all rows from both tables, including unmatched rows from either table.

```
SELECT *
FROM products
FULL OUTER JOIN sales ON products.product_id = sales.product_id;
```

- `LEFT OUTER JOIN` **Statement**: This join returns all rows from the left table and matching rows from the right table.

```
SELECT *
FROM products
LEFT JOIN sales ON products.product_id = sales.product_id;
```

- `RIGHT OUTER JOIN` **Statement**: This join returns all rows from the right table and matching rows from the left table.

```
SELECT *
FROM products
RIGHT JOIN sales ON products.product_id = sales.product_id;
```

- `SELF JOIN` **Statement**: This join is used to join a table with itself to compare rows within the same table.

```
SELECT A.column_name, B.column_name
FROM table_name A
JOIN table_name B ON A.related_column = B.related_column;
```

- `CROSS JOIN` **Statement**: This join returns the Cartesian product of two tables, combining every row from the first table with every row from the second table.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

PostgreSQL Specific Commands

- `psql` **Login Command**: This command is used to log in to PostgreSQL with specific username and database.

```
psql --username=freecodecamp --dbname=postgres
```

- `\l` **Command**: This command lists all databases in the PostgreSQL instance.

```
\l
```

- **`CREATE DATABASE` and `\c` Commands**: These commands are used to create a new database and connect to it.

```
CREATE DATABASE database_name;
\c database_name
```

- **`\d` Command**: This command lists all tables in the current database.

```
\d
```

- **`\d table_name` Command**: This command displays the schema/structure of a specific table.

```
\d table_name
```

- **`\q` Command**: This command exits the PostgreSQL client.

```
\q
```

Relational vs Non-Relational

- **Non-Relational (NoSQL) Databases**: Store unstructured or semi-structured data. Do not require a rigid schema and are more flexible for evolving data models.
- **Choosing Between Relational and Non-Relational**: Depends on the nature of your data and application requirements.
- **Relational vs Non-Relational**: Choose relational for structured data and consistency; NoSQL for flexibility and fast-changing data.

Popular RDBMS Systems

- **MySQL**: Open-source, reliable, widely used in web development, supported by a large community.
- **PostgreSQL**: Open-source, advanced, extensible. Supports custom data types and server-side programming.
- **SQLite**: Lightweight, file-based, serverless. Ideal for small applications.
- **Microsoft SQL Server**: Proprietary, enterprise-grade database.
- **Oracle Database**: Commercial RDBMS known for large-scale performance and scalability.

Best Practices

- **Naming Convention**: Use `snake_case` (e.g., `delivery_orders`) for table and column names.

Bash Scripting Basics

- **Bash scripting**: Writing a sequence of Bash commands in a file, which you can then execute with Bash to run the contents of the file.

- **Shebang**: The commented line at the beginning of a script (e.g., `#!/bin/bash`) that indicates what interpreter should be used for the script.

  ```
  #!/bin/bash
  ```

- **Variable assignment**: Instantiate variables using the syntax `variable_name=value`.

```
servers=("prod" "dev")
```

- **Variable creation rules**: Create variables with `VARIABLE_NAME=VALUE` syntax. No spaces are allowed around the equal sign (`=`). Use double quotes if the value contains spaces.

```
NAME=John
MESSAGE="Hello World"
COUNT=5
TEXT="The next number is, "
```

- **Variable usage**: Access variable values by placing `$` in front of the variable name.

```
echo $NAME
echo "The message is: $MESSAGE"
```

- **Variable interpolation**: Use `$variable_name` to access the value of a variable within strings and commands.

```
TEXT="The next number is, "
NUMBER=42
echo $TEXT B:$NUMBER
echo $TEXT I:$NUMBER

echo "Pulling $server"
rsync --archive --verbose $server:/etc/nginx/conf.d/server.conf
configs/$server.conf
```

- **Variable scope**: Shell scripts run from top to bottom, so variables can only be used below where they are created.

```
NAME="Alice"
echo $NAME
```

- **User input**: Use `read` to accept input from users and store it in a variable.

```
read USERNAME
echo "Hello $USERNAME"
```

- **Comments**: Add comments to your scripts using `#` followed by your comment text.
  - Single-line comments start with `#` and continue to the end of the line
  - Comments are ignored by the shell and don't affect script execution

```
# This is a single-line comment
NAME="John"  # Comment at end of line
```

- **Multi-line comments**: Comment out blocks of code using colon and quotes.

```
: '
This is a multi-line comment
Everything between the quotes is ignored
Useful for debugging or documentation
'
```

- **Built-in commands and help**:
  - Use `help` to see a list of built-in bash commands
  - Use `help <command>` to get information about specific built-in commands
  - Some commands (like `if`) are built-ins and don't have man pages
  - Built-in commands are executed directly by the shell rather than as external programs
  - Use `help function` to see information about creating functions

```
help
help if
help function
```

- **Finding command locations**: Use `which` to locate where executables are installed.
  - Shows the full path to executable files
  - Useful for finding interpreter locations (like bash)
  - Helps verify which version of a command will be executed

```
which bash
which python
which ls
```

- **Manual pages**: Use `man` to access detailed documentation for commands.
  - Provides comprehensive information about command usage
  - Shows all available options and examples
  - Use arrow keys to navigate, 'q' to quit
  - Not all commands have manual pages (built-ins use `help` instead)

```
man echo
man ls
man bash
```

- **Help flags**: Many commands support `--help` for quick help information.
  - Alternative to manual pages for quick reference
  - Shows command syntax and common options
  - Not all commands support this flag (some may show error)

```
ls --help
chmod --help
mv --help
```

- **Echo command options**: The `echo` command supports various options:

  - `-e` option enables interpretation of backslash escapes
  - `\n` creates a new line
  - Empty lines are only printed when values are enclosed in quotes
  - Useful for creating formatted output and program titles

```
echo -e "Line 1\nLine 2"
echo ""
echo -e "\n~~ Program Title ~~\n"
echo "Line 1\nLine 2"
```

- **Script arguments**: Programs can accept arguments that are accessible using `$` variables.
  - `$*` prints all arguments passed to the script
  - `$@` prints all arguments passed to the script as separate quoted strings
  - `$<number>` accesses specific arguments by position (e.g., `$1`, `$2`, `$3`)

```
echo $*
echo $@
echo $1
echo $2
```

Double Bracket Expressions `[[ ]]`

- **Double bracket syntax**: Use `[[ ]]` for conditional testing and pattern matching.
  - Must have spaces inside the brackets and around operators
  - Returns exit status 0 (true) or 1 (false) based on the test result

```
[[ $variable == "value" ]]
[[ $number -gt 10 ]]
[[ -f filename.txt ]]
```

- **String comparison operators**: Compare strings using various operators within `[[ ]]`.
  - `==` (equal): Tests if two strings are identical
  - `!=` (not equal): Tests if two strings are different
  - `<` (lexicographically less): String comparison in alphabetical order
  - `>` (lexicographically greater): String comparison in alphabetical order

```
[[ "apple" == "apple" ]]
[[ "apple" != "orange" ]]
[[ "apple" < "banana" ]]
[[ "zebra" > "apple" ]]
```

- **Numeric comparison operators**: Compare numbers using specific numeric operators.
  - `-eq` (equal): Numeric equality comparison
  - `-ne` (not equal): Numeric inequality comparison
  - `-lt` (less than): Numeric less than comparison
  - `-le` (less than or equal): Numeric less than or equal comparison
  - `-gt` (greater than): Numeric greater than comparison
  - `-ge` (greater than or equal): Numeric greater than or equal comparison

```
```

```
[[ $number -eq 5 ]]
[[ $count -ne 0 ]]
[[ $age -ge 18 ]]
[[ $score -lt 100 ]]
```

- **Logical operators**: Combine multiple conditions using logical operators.
  - `&&` (and): Both conditions must be true
  - `||` (or): At least one condition must be true
  - `!` (not): Negates the condition (makes true false, false true)

```
[[ $age -ge 18 && $age -le 65 ]]
[[ $name == "John" || $name == "Jane" ]]
[[ ! -f missing_file.txt ]]
```

- **File test operators**: Test file properties and existence.
  - `-e file`: True if file exists
  - `-f file`: True if file exists and is a regular file
  - `-d file`: True if file exists and is a directory
  - `-r file`: True if file exists and is readable
  - `-w file`: True if file exists and is writable
  - `-x file`: True if file exists and is executable
  - `-s file`: True if file exists and has size greater than zero

```
[[ -e /path/to/file ]]
[[ -f script.sh ]]
[[ -d /home/user ]]
[[ -x program ]]
```

- **Pattern matching with `=~`**: Use regular expressions for advanced pattern matching.
  - `=~` operator enables regex pattern matching
  - Pattern should not be quoted when using regex metacharacters
  - Supports full regular expression syntax
  - Case-sensitive by default

```
[[ "hello123" =~ [0-9]+ ]]
[[ "email@domain.com" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
]]
[[ "filename.txt" =~ \.txt$ ]]
```

- **Variable existence testing**: Check if variables are set or empty.
  - Test if variable is empty: `[[ ! $variable ]]`

```
[[ ! $undefined_var ]]
```

Double Parentheses Expressions `(( ))`

- **Arithmetic evaluation**: Use `(( ))` for mathematical calculations and numeric comparisons.
  - Evaluates arithmetic expressions using C-style syntax

- Variables don't need `$` prefix inside double parentheses
- Returns exit status 0 if result is non-zero, 1 if result is zero
- Supports all standard arithmetic operators

```
(( result = 10 + 5 ))
(( count++ ))
(( total += value ))
```

- **Arithmetic operators**: Mathematical operators available in `(( ))`.
  - `+` (addition): Add two numbers
  - `-` (subtraction): Subtract second number from first
  - `*` (multiplication): Multiply two numbers
  - `/` (division): Divide first number by second (integer division)
  - `%` (modulus): Remainder after division
  - `**` (exponentiation): Raise first number to power of second

```
(( sum = a + b ))
(( diff = x - y ))
(( product = width * height ))
(( remainder = num % 10 ))
(( power = base ** exponent ))
```

- **Assignment operators**: Modify variables using arithmetic assignment operators.
  - `=` (assignment): Assign value to variable
  - `+=` (add and assign): Add value to variable
  - `-=` (subtract and assign): Subtract value from variable
  - `*=` (multiply and assign): Multiply variable by value
  - `/=` (divide and assign): Divide variable by value
  - `%=` (modulus and assign): Set variable to remainder

```
(( counter = 0 ))
(( counter += 5 ))
(( total -= cost ))
(( area *= 2 ))
(( value /= 3 ))
```

- **Increment and decrement operators**: Modify variables by one.
  - `++variable` (pre-increment): Increment before use
  - `variable++` (post-increment): Increment after use
  - `--variable` (pre-decrement): Decrement before use
  - `variable--` (post-decrement): Decrement after use

```
(( ++counter ))
(( index++ ))
(( --remaining ))
(( attempts-- ))
```

- **Comparison operators**: Compare numbers using arithmetic comparison.
  - `==` (equal): Numbers are equal
  - `!=` (not equal): Numbers are not equal

- `<` (less than): First number is less than second
- `<=` (less than or equal): First number is less than or equal to second
- `>` (greater than): First number is greater than second
- `>=` (greater than or equal): First number is greater than or equal to second

```
(( age >= 18 ))
(( score < 100 ))
(( count == 0 ))
(( temperature > freezing ))
```

- **Logical operators**: Combine arithmetic conditions.
  - `&&` (and): Both conditions must be true
  - `||` (or): At least one condition must be true
  - `!` (not): Negates the condition

```
(( age >= 18 && age <= 65 ))
(( score >= 90 || extra_credit > 0 ))
(( !(count == 0) ))
```

- **Bitwise operators**: Perform bit-level operations on integers.
  - `&` (bitwise AND): AND operation on each bit
  - `|` (bitwise OR): OR operation on each bit
  - `^` (bitwise XOR): XOR operation on each bit
  - `~` (bitwise NOT): Invert all bits
  - `<<` (left shift): Shift bits to the left
  - `>>` (right shift): Shift bits to the right

```
(( result = a & b ))
(( flags |= new_flag ))
(( shifted = value << 2 ))
```

- **Conditional (ternary) operator**: Use `condition ? true_value : false_value` syntax.
  - Provides a concise way to assign values based on conditions
  - Similar to the ternary operator in C-style languages
  - Evaluates condition and returns one of two values

```
(( result = (score >= 60) ? 1 : 0 ))
(( max = (a > b) ? a : b ))
(( sign = (num >= 0) ? 1 : -1 ))
```

- **Command substitution with arithmetic**: Use `$(( ))` to capture arithmetic results.
  - Returns the result of the arithmetic expression as a string
  - Can be used in assignments or command arguments
  - Useful for calculations that need to be used elsewhere

```
result=$(( 10 + 5 ))
echo "The answer is $(( a * b ))"
array_index=$(( RANDOM % array_length ))
```

Control Flow and Conditionals

- **Conditional statements**: Use `if` statements to execute code based on conditions.
  - Basic syntax: `if [[ CONDITION ]] then STATEMENTS fi`
  - Full syntax: `if [[ CONDITION ]] then STATEMENTS elif [[ CONDITION ]] then STATEMENTS else STATEMENTS fi`
  - Can use both `[[ ]]` and `(( ))` expressions for different types of conditions
  - **elif (else if)**: Optional, can be repeated multiple times to test additional conditions in sequence
  - **else**: Optional, executes when all previous conditions are false
  - Can mix double parentheses `(( ... ))` and double brackets `[[ ... ]]` in same conditional chain

```
if (( NUMBER <= 15 ))
then
    echo "B:$NUMBER"
elif [[ $NUMBER -le 30 ]]
then
    echo "I:$NUMBER"
elif (( NUMBER < 46 ))
then
    echo "N:$NUMBER"
elif [[ $NUMBER -lt 61 ]]
then
    echo "G:$NUMBER"
else
    echo "O:$NUMBER"
fi
```

Command Execution and Process Control

- **Command separation**: Use semicolon (`;`) to run multiple commands on a single line.
  - Commands execute sequentially from left to right
  - Each command's exit status can be checked individually

```
[[ 4 -ge 5 ]]; echo $?
ls -l; echo "Done"
```

- **Exit status**: Every command has an exit status that indicates success or failure.
  - Access exit status of the last command with `$?`
  - Exit status `0` means success (true/no errors)
  - Any non-zero exit status means failure (false/errors occurred)
  - Common error codes: `127` (command not found), `1` (general error)

```
echo $?
[[ 4 -le 5 ]]; echo $?
ls; echo $?
bad_command; echo $?
```

- **Subshells and command substitution**: Different uses of parentheses for execution contexts.
  - Single parentheses `( ... )` create a subshell
  - `$( ... )` performs command substitution

- Subshells run in separate environments and don't affect parent shell variables

```
( cd /tmp; echo "Current dir: $(pwd)" )
current_date=$(date)
file_count=$(ls | wc -l)
echo "Today is $current_date"
echo "Found $file_count files"
```

- **Sleep command**: Pause script execution for a specified number of seconds.
  - Useful for creating delays in scripts
  - Can be used with decimal values for subsecond delays

```
sleep 3
sleep 0.5
sleep 1
```

Loops

- **While loops**: Execute code repeatedly while a condition is true.
  - Syntax: `while [[ CONDITION ]] do STATEMENTS done`

```
I=5
while [[ $I -ge 0 ]]
do
    echo $I
    (( I-- ))
    sleep 1
done
```

- **Until loops**: Execute code repeatedly until a condition becomes true.
  - Syntax: `until [[ CONDITION ]] do STATEMENTS done`

```
until [[ $QUESTION =~ \?$ ]]
do
    echo "Please enter a question ending with ?"
    read QUESTION
done
until [[ $QUESTION =~ \?$ ]]
do
    GET_FORTUNE again
done
```

- **For loops**: Iterate through arrays or lists using `for` loops with `do` and `done` to define the loop's logical block.

```
for server in "${servers[@]}"
do
    echo "Processing $server"
done
for (( i = 1; i <= 5; i++ ))
do
    echo "Number: $i"
```

```
done
for (( i = 5; i >= 1; i-- ))
do
    echo "Countdown: $i"
done
for i in {1..5}
do
    echo "Count: $i"
done
```

Arrays

- **Arrays**: Store multiple values in a single variable.
  - Create arrays with parentheses: `ARRAY=("value1" "value2" "value3")`
  - Access elements by index: `${ARRAY[0]}`, `${ARRAY[1]}`
  - Access all elements: `${ARRAY[@]}` or `${ARRAY[*]}`
  - Array indexing starts at 0

```
RESPONSES=("Yes" "No" "Maybe" "Ask again later")

echo ${RESPONSES[0]}     # Yes
echo ${RESPONSES[1]}     # No
echo ${RESPONSES[5]}     # Index 5 doesn't exist; empty string
echo ${RESPONSES[@]}     # Yes No Maybe Ask again later
echo ${RESPONSES[*]}     # Yes No Maybe Ask again later
```

- **Array inspection with declare**: Use `declare -p` to view array details.
  - Shows the array type with `-a` flag
  - Displays all array elements and their structure

```
ARR=("a" "b" "c")
declare -p ARR # ARR=([0]="a" [1]="b" [2]="c")
```

- **Array expansion**: Use `"${array_name[@]}"` syntax to expand an array into individual elements.

```
for server in "${servers[@]}"
```

Functions

- **Functions**: Create reusable blocks of code.
  - Define with `FUNCTION_NAME() { STATEMENTS }`
  - Call by using the function name
  - Can accept arguments accessible as `$1`, `$2`, etc.

```
GET_FORTUNE() {
    echo "Ask a question:"
    read QUESTION
}
GET_FORTUNE
```

- **Function arguments**: Functions can accept arguments just like scripts.

- Arguments are passed when calling the function
- Access arguments inside function using `$1`, `$2`, etc.
- Use conditional logic to handle different arguments

```
GET_FORTUNE() {
    if [[ ! $1 ]]
    then
        echo "Ask a yes or no question:"
    else
        echo "Try again. Make sure it ends with a question mark:"
    fi
    read QUESTION
}
GET_FORTUNE
GET_FORTUNE again
```

Random Numbers and Mathematical Operations

- **Random numbers**: Generate random values using the `$RANDOM` variable.
  - `$RANDOM` generates numbers between 0 and 32767
  - Use modulus operator to limit range: `$RANDOM % 75`
  - Add 1 to avoid zero: `$(( RANDOM % 75 + 1 ))`
  - Must use `$(( ... ))` syntax for calculations with `$RANDOM`

```
NUMBER=$(( RANDOM % 6 ))
DICE=$(( RANDOM % 6 + 1 ))
BINGO=$(( RANDOM % 75 + 1 ))
echo $(( RANDOM % 10 ))
```

- **Random array access**: Use random numbers to access array elements randomly.
  - Generate random index within array bounds
  - Use random index to access array elements
  - Useful for random selections from predefined options

```
RESPONSES=("Yes" "No" "Maybe" "Outlook good" "Don't count on it" "Ask
again later")
N=$(( RANDOM % 6 ))
echo ${RESPONSES[$N]}
```

- **Modulus operator**: Use `%` to get the remainder of division operations.
  - Essential for limiting random number ranges
  - Works with `$RANDOM` to create bounded random values
  - `RANDOM % n` gives numbers from 0 to n-1

```
echo $(( 15 % 4 ))
echo $(( RANDOM % 100 ))
echo $(( RANDOM % 10 + 1 ))
```

Environment and System Information

- **Environment variables**: Predefined variables available in the shell environment.

  - `$RANDOM`: Generates random numbers between 0 and 32767
  - `$LANG`: System language setting
  - `$HOME`: User's home directory path
  - `$PATH`: Directories searched for executable commands
  - View all with `printenv` or `declare -p`

```
echo $RANDOM
echo $HOME
echo $LANG
printenv
```

- **Variable inspection**: Use `declare` to view and work with variables.
  - `declare -p`: Print all variables and their values
  - `declare -p VARIABLE`: Print specific variable details
  - Shows variable type (string, array, etc.) and attributes

```
declare -p
declare -p RANDOM
declare -p MY_ARRAY
```

- **Command types**: Different categories of commands available in bash.
  - **Built-in commands**: Executed directly by the shell (e.g., `echo`, `read`, `if`)
  - **External commands**: Binary files in system directories (e.g., `ls`, `sleep`, `bash`)
  - **Shell keywords**: Language constructs (e.g., `then`, `do`, `done`)
  - Use `type <command>` to see what type a command is

```
type echo
type ls
type if
type ./script.sh
```

File Creation and Management

- **File creation**: Use `touch` to create new empty files.
  - Creates a new file if it doesn't exist
  - Updates the timestamp if the file already exists
  - Commonly used to create script files before editing

```
touch script.sh
touch bingo.sh
touch filename.txt
```

Creating and Running Bash Scripts

- **Script execution methods**: Multiple ways to run bash scripts:
  - **`sh scriptname.sh`**: Run with the sh shell interpreter.
  - **`bash scriptname.sh`**: Run with the bash shell interpreter.
  - **`./scriptname.sh`**: Execute directly (requires executable permissions).

```

```

```
sh questionnaire.sh
bash questionnaire.sh
./questionnaire.sh
```

File Permissions and Script Execution

- **Permission denied error**: When using `./scriptname.sh`, you may get "permission denied" if the file lacks executable permissions.
- **Checking permissions**: Use `ls -l` to view file permissions.

  ```
  ls -l questionnaire.sh
  ```

- **Permission format**: The output shows permissions as `-rw-r--r--` where:
  - First character (`-`): File type (- for regular file, d for directory)
  - Next 9 characters: Permissions for owner, group, and others
  - `r` = read, `w` = write, `x` = execute
- **Adding executable permissions**: Use `chmod +x` to give executable permissions to everyone.

  ```
  chmod +x questionnaire.sh
  ```

- **Script organization**: Best practices for structuring bash scripts.
  - Start with shebang (`#!/bin/bash`)
  - Add descriptive comments about script purpose
  - Define variables at the top
  - Group related functions together
  - Main script logic at the bottom

  ```
  #!/bin/bash
  NAME="value"
  ARRAY=("item1" "item2")
  my_function() {
      echo "Function code here"
  }
  my_function
  echo "Script complete"
  ```

- **Sequential script execution**: Create master scripts that run multiple programs in sequence.
  - Useful for automating workflows that involve multiple scripts
  - Each script runs to completion before the next one starts
  - Can combine different programs into a single execution flow
  - Arguments can be passed to individual scripts as needed
  - Can include different types of programs (interactive, automated, etc.)

  ```
  #!/bin/bash
  ./setup.sh
  ./interactive.sh
  ./processing.sh
  ./cleanup.sh
  ```

Database Normalization

This is the process of organizing a relational database to reduce data redundancy and improve integrity.

Its benefits include:

- Minimizing duplicated data, which saves storage and reduces inconsistencies.
- Enforcing data integrity through the use of primary and foreign keys.
- Making databases easier to maintain and understand.

Normal Forms

- **1NF (First Normal Form)**
  - Each cell contains a single (atomic) value.
  - Each record is unique (enforced by a primary key).
  - Order of rows/columns is irrelevant.
  - Example: Move multiple phone numbers from a `students` table into a separate `student_phones` table.
- **2NF (Second Normal Form)**
  - Meets 1NF requirements.
  - No **partial dependencies**: every non-key attribute must depend on the entire composite primary key.
  - Example: Split `orders` table into `order_header` and `order_items` to avoid attributes depending on only part of the key.
- **3NF (Third Normal Form)**
  - Meets 2NF requirements.
  - No **transitive dependencies**: non-key attributes cannot depend on other non-key attributes.
  - Example: Move `city_postal_code` to a `cities` table instead of storing it with every order.
- **BCNF (Boyce-Codd Normal Form)**
  - Meets 3NF requirements.
  - Every determinant (left-hand side of a functional dependency) must be a superkey.

Tip: Aim for 3NF in most designs for a good balance of integrity and performance.

Key SQL Concepts

- SQL is a Structured Query Language for communicating with relational databases.
- **Basic commands** → `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, `ALTER TABLE`, etc.
- `Joins` → Combines data from multiple tables (`INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`).

Running SQL Commands in Bash

You can run SQL commands directly from the command line using the `psql` command-line client for PostgreSQL or similar tools for other databases.

For example, to run a SQL file in PostgreSQL:

```
psql -U username -d database_name -c "SELECT * FROM students;"
```

You can also execute MySQL commands directly:

```
mysql -u username -p database_name -e "SELECT * FROM students;"
```

Run SQL from a File

```
# PostgreSQL
psql -U username -d database_name -f script.sql

# MySQL
mysql -u username -p database_name < script.sql
```

Embed SQL in a Bash Script

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Insert student data
psql -U "$DB_USER" -d "$DB_NAME" -c \
"INSERT INTO students (name, age, major) VALUES ('Alice', 20, 'CS');"
```

Use of Variables in SQL

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"
STUDENT_NAME="Bob"
AGE=21

psql -U "$DB_USER" -d "$DB_NAME" -c \
"INSERT INTO students (name, age) VALUES ('$STUDENT_NAME', $AGE);"
```

Tip: Sanitize variables to avoid SQL injection.

Retrieving and Using SQL Query Results in Bash

When you run SQL queries via `psql`, you can **capture** and **process** the returned values in your Bash scripts.

Capturing a Single Value

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get total student count
STUDENT_COUNT=$(psql -U "$DB_USER" -d "$DB_NAME" -t -A -c \
"SELECT COUNT(*) FROM students;")

echo "Total students: $STUDENT_COUNT"
```

Output → 42

Retrieving Multiple Columns

```
#!/bin/bash
DB_USER="school_admin"
```

```
DB_NAME="school"

# Get top 3 students' names and ages
RESULTS=$(psql -U "$DB_USER" -d "$DB_NAME" -t -A -F"," -c \
"SELECT name, age FROM students LIMIT 3;")

echo "Top 3 students:"
echo "$RESULTS"
```

Output

```
Alice,20
Bob,21
Charlie,22
```

Looping Through Query Results

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get student names and majors
psql -U "$DB_USER" -d "$DB_NAME" -t -A -F"," -c \
"SELECT name, major FROM students;" | while IFS="," read -r name major
do
    echo "Student: $name | Major: $major"
done
```

Shape of Output

```
Student: Alice | Major: CS
Student: Bob   | Major: Math
Student: Carol | Major: Physics
```

SQL Injection

It is a web security vulnerability where attackers insert malicious SQL code into input fields to manipulate the database.

This can lead to risky actions like:

- Bypassing authentication.
- Stealing sensitive data.
- Modifying or deleting records.

An example of an SQL injection attack:

```
SELECT * FROM users WHERE username = ' " " OR "1"="1" -- ' AND password =
'anything';
```

This query would return all users because the condition `OR "1"="1"` is always true, allowing attackers to bypass login checks.

Preventing SQL Injection

1. **Use Prepared Statements**: These separate SQL code from data, preventing injection. Here's an example (Node.js with pg):

   ```
   client.query('SELECT * FROM users WHERE username = $1 AND password = $2',
   [username, password]);
   ```

2. **Input Validation**: Sanitize and validate all user inputs to ensure they conform to expected formats.

3. **Least Privilege**: Use database accounts with the minimum permissions necessary for the application.

Note: Never grant admin rights to application accounts.

N+1 Problem

The N+1 problem occurs when an application makes one query to retrieve a list of items (N) and then makes an additional query for each item to retrieve related data, resulting in N+1 queries.

Why It's Bad

- Each query adds network and processing overhead.
- Multiple small queries are slower than one optimized query.

Example of N+1 Pattern

```
-- 1: Get list of orders
SELECT * FROM orders LIMIT 50;

-- N: For each order, get customer
SELECT * FROM customers WHERE customer_id = ...;
```

Solution: Use `JOINs` or other set-based operations.

```
SELECT
  orders.order_id,
  orders.product,
  orders.quantity,
  customers.customer_id,
  customers.name,
  customers.email,
  customers.address
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id
WHERE orders.order_id IN (SELECT order_id FROM orders LIMIT 50);
```

Always look for opportunities to combine related data into a single query.

Introduction to Version Control

- **Definition**: A version control system allows you to track and manage changes in your project. Examples of version control systems used in software are Git, SVN, or Mercurial.

Cloud-Based Version Control Providers

- **List of Cloud-Based Version Control Providers**: GitHub and GitLab are popular examples of cloud-based version control providers that allow software teams to collaborate and manage repositories.

Installing and Setting up Git

- **Installing Git**: To check if Git is already installed on your machine you can run the following command in the terminal:

```
git --version
```

If you see a version number, that means Git is installed. If not, then you will need to install it.

For Linux systems, Git often comes preinstalled with most distros. If you do not have Git pre-installed, you should be able to install it with your package manager commands such as `sudo apt-get install git` or `sudo pacman -S git`.

For Mac users, you can install Git via Homebrew with `brew install git`, or you can download the executable installer from Git's website.

For Windows, you can download the executable installer from Git's website. Or, if you have set up Chocolatey, you can run `choco install git.install` in PowerShell. Note that on Windows, you may also want to download Git Bash so you have a Unix-like shell environment available.

To make sure the installation worked, run the `git --version` command again in the terminal.

- **Git Configurations**: `git config` is used to set configuration variables that are responsible for how Git operates on your machine. To view your current setting variables and where they are stored on your system, you can run the following command:

```
git config --list --show-origin
```

Right now you should be seeing only system-level configuration settings if you just installed Git for the first time.

To set your user name, you can run the following command:

```
git config --global user.name "Jane Doe"
```

The `--global` flag is used here to set the user name for all projects on your system that use Git. If you need to override the user name for a particular project, then you can run the command in that particular project directory without the `--global` flag.

To set the user email address, you can run the following command:

```
git config --global user.email janedoe@example.com
```

Another configuration you can set is the preferred editor you want Git to use. Here is an example of how to set your preferred editor to Emacs:

```
git config --global core.editor emacs
```

If you choose not to set a preferred editor, then Git will default to your system's default editor.

Open vs. Closed Source Software

- **Definition**: "Open-source" means people can see the code you publish, propose changes, report issues, and even run a modified version. "Closed-source" means the only people who can see and interact with the project are the people you explicitly authorize.

GitHub

- **Definition**: GitHub is a cloud-based solution that offers storage of version-controlled projects in something called "repositories", and enables collaboration features to use with those projects.
- **GitHub CLI**: This tool is used to do GitHub-specific tasks without leaving the command line. If you do not have it installed, you can get instructions to do so from GitHub's documentation - but you should have it available in your system's package manager.
- **GitHub Pages**: GitHub Pages is an option for deploying static sites, or applications that do not require a back-end server to handle logic. That is, applications that run entirely client-side, or in the user's browser, can be fully deployed on this platform.
- **GitHub Actions**: GitHub Actions is a feature that lets you automate workflows directly in your GitHub repository including building, testing, and deploying your code.

Common Git Commands

- `git init`: This will initialize an empty Git repository so Git can begin tracking changes for this project. When you initialize an empty Git repository to a project, a new `.git` hidden directory will be added. This `.git` directory contains important information for Git to manage your project.
- `git status`: This command is used to show the current state of your working directory - you will be using this command a lot in your workflow.
- `git add`: This command is used to stage your changes. Anything in the staging area will be added for the next commit. If you want to stage all unstaged changes, then you can use `git add .` The period (`.`) is an alias for the current directory you are in.
- `git commit`: This command is used to commit your changes. A commit is a snapshot of your project state at that given time. If you run `git commit`, it will open up your preferred editor you set in the Git configuration. Once the editor is open, you can provide a detailed message of your changes. You can also choose to provide a shorter message by using the `git commit -m` command like this:

```
git commit -m "short message goes here"
```

- `git log`: This will list all prior commits with helpful information like the author, date of commit, commit message and commit hash. The commit hash is a long string which serves as a unique identifier for a commit.
- `git remote add`: This command is used to setup the remote connection to your remote repo.
- `git push`: This command is used to push up your changes to a remote repository.
- `git pull`: This command is used to pull down the latest changes from your remote repository into your local repository.
- `git clone`: This command will clone a repository. This means you will have a copy of the repository. This copy includes the repository history, all files/folders and commits on your local device.
- `git remote -v`: This command will show the list of remote repositories associated with your local Git repository.
- `git branch`: This command will list all of your local branches.
- `git fetch upstream`: This command tells Git to go get the latest changes that are on your upstream remote (which is the original repo).
- `git merge upstream/main`: This command tells Git to merge the latest changes from the `main` branch in the upstream remote into your current branch.
- `git reset`: This command allows you to reset the current state of a branch. Passing the `--hard` flag tells Git to force the local files to match the branch state. This ensures that you have a clean slate to work from.
- `git rebase`: A rebase in Git is a way to move or combine a sequence of commits from one branch onto another.

Working with Branches

- **Definition**: A branch in Git is a separate workspace where you can make changes. The `main` branch will often represent the primary or production branch in a real world application. Developer teams will create multiple branches for new features and bug fixes and then merge those changes back into the `main` branch.
- **Creating a New Branch**: To create a new branch you can run the following command:

```
git branch feature
```

To checkout that branch, you can run the following command:

```
git checkout feature
```

Most developers will use the shorthand command for creating and checking out a branch which is the following:

```
git checkout -b new-branch-name
```

A newer and alternative command would be the `git switch` command. Here is an example for creating and switching to a new branch:

```
git switch -c new-branch-name
```

- **Branching Strategies**: Your `main` branch is your default branch and usually is pretty stable. So it is best to branch off from there to create new branches for items like bug fixes, new features, or other miscellaneous work.
- **Merge Conflicts**: This happens when Git tries to automatically merge changes from different branches but can't decide which changes to keep. This usually happens when there are conflicting changes for the same portion of the file.

Five States for a Git Tracked File

- **"Untracked"**: This means that the file is new to the repository, and Git has not "seen" it before.
- **"Modified"**: This file existed in the previous commit, and has changes that have not been committed.
- **"Ignored"**: You likely won't see ignored files in Git, but your IDE might have an indicator for them. Ignored files are excluded from Git operations, typically because they are included in the `.gitignore` file.
- **"Deleted"**: A deleted file is the opposite of an untracked file - it's a file that previously existed, and has been removed.
- **"Renamed"**: A renamed file is a file where the contents are unchanged, but the name or location of the file was modified. In some cases, a file can be considered renamed even if it has a small amount of changes.

`.gitignore` Files

- **Definition**: The `.gitignore` file is a special type of file related to Git operations. The name suggests that this file is used to tell Git to ignore things, and that's the common use case. But what it actually does is it tells Git to stop tracking a file.

Working with Repositories

- **Definition**: A repository is like a container for a project - if you are working on an app, you would keep the files for that app together in a repository. Repositories can be local on your computer, or remote on a service like GitHub.
- **Public vs. Private Repositories**: A public repository can be viewed and downloaded by anyone. A private repository can only be accessed by you, and anyone you grant explicit access to.
- **Creating Repositories on GitHub**: To create a new repository on GitHub, you can click on the `"New Repository"` button and walk through the GitHub UI of setting up a new repository.
- **Pushing Local Repositories to GitHub**: If you have a local project on your computer, you can push up that repository to GitHub. Here is a step-by-step overview of the process:

1. Initialize an empty git repository in the project directory (`git init`).
2. Make changes to your project.
3. Run the `git status` command to see all changes made that are being tracked by git.
4. Stage your changes (`git add`).
5. Commit your changes (`git commit`).
6. Setup the remote connection (`git remote add`).
7. Push your changes to GitHub (`git push`).

Pull Requests

- **Pull Requests**: A pull request is a request to pull changes in from your branch into the target branch. Pull requests are the flow you use when you want to contribute code changes to a project. This approach allows the maintainers of the project to review your changes. They can leave comments, ask questions, and suggest tweaks. Then once the review process is complete, it can be approved and merged into the main branch.

Contributing to Other Repositories

- **Process**: There are thousands of projects that you can contribute to. Here is the basic process on how to contribute to another repository:

1. Read the contributing documentation
2. Find an available issue to work on
3. Fork the repository
4. Clone your forked copy of the repository
5. Create a new branch
6. Make the changes according to the issue
7. Create a PR (Pull Request)
8. Wait for a review for that PR

Working with SSH and GPG Keys

- **GPG Keys**: GPG, or Gnu Privacy Guard, keys are typically used to sign files or commits. Someone can then use your public GPG key to verify that the file signature is from your key and that the contents of the file have not been modified or tampered with.

To generate a GPG key, you'll need to run:

```
gpg --full-generate-key
```

- **SSH Keys**: SSH, or Secure SHell, keys are typically used to authenticate a remote connection to a server - via the `ssh` utility. You can also use an SSH key to sign commits.

For an SSH key, you'll run:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

`ed25519` is a modern public-key signature algorithm.

- **Signing Commits with GPG Keys**: In order to sign your commits with your GPG key, you'll need to upload your public key, not the private key, to your GitHub account. To list your public keys, you will need to run the following:

```
gpg --list-secret-keys --keyid-format=long
```

Then, to get the public key, use:

```
gpg --armor --export "<key id>"
```

Then, take the short ID you got from listing the keys and run this command to set it as your git signing key:

```
git config --global user.signingkey <your_gpg_key_id>
```

Then, you can pass the `-S` flag to your `git commit` command to sign a specific commit - you'll need to provide your passphrase. Alternatively, if you want to sign every commit automatically, you can set the autosign config to `true`:

```
git config --global commit.gpgsign true
```

- **Signing Commits with SSH Keys**: To sign with an SSH key, which is a relatively new feature on GitHub, you'll need to start by uploading the key to your GitHub account. Then you'll need to set the signing mode for git to use SSH:

```
git config --global gpg.format ssh
```

Then, to set the signing key, you'll pass the file path instead of an ID:

```
git config --global user.signingkey <path_to_your_ssh_keys>
```

--assignment--

Review Bash, SQL and other Relational Database topics and concepts.

Responsive Web Design Review

Responsive Web Design

- **Definition**: The core principle of responsive design is adaptability – the ability of a website to adjust its layout and content based on the screen size and capabilities of the device it's being viewed on.
- **Fluid grids**: These use relative units like percentages instead of fixed units like pixels, allowing content to resize and reflow based on screen size.
- **Flexible images**: These are set to resize within their containing elements, ensuring they don't overflow their containers on smaller screens.

Media Queries

- **Definition**: This allows developers to apply different styles based on the characteristics of the device, primarily the viewport width.

```
@media screen and (min-width: 768px) {
  /* Styles for screens at least 768px wide */
}
```

- `all` **Media Type**: This is suitable for all devices. This is the default if no media type is specified.
- `print` **Media Types**: This is intended for paged material and documents viewed on a screen in print preview mode.
- `screen` **Media Types**: This is intended primarily for screens.
- `aspect-ratio`: This describes the ratio between the width and height of the viewport.

```
@media screen and (aspect-ratio: 16/9) {
  /* Styles for screens with a 16:9 aspect ratio */
}
```

- `orientation`: This is used to indicate whether the device is in landscape or portrait orientation.

```
@media screen and (orientation: landscape) {
  /* Styles for landscape orientation */
}
```

- `resolution`: This is used to describe the resolution of the output device in dots per inch (dpi) or dots per centimeter (dpcm).

```
@media screen and (min-resolution: 300dpi) {
  /* Styles for high-resolution screens */
}
```

- `hover`: This is used to test whether the primary input mechanism can hover over elements.

```
@media (hover: hover) {
  /* Styles for devices that support hover */
}
```

- `prefers-color-scheme`: This is used to detect if the user has requested a light or dark color theme.
- **Media Queries and Logical Operators**: The `and` operator is used to combine multiple media features, while `not` and `only` can be used to negate or isolate media queries.

```
@media screen and (min-width: 768px) and (orientation: landscape) {
  /* Styles for landscape screens at least 768px wide */
}
```

Common Media Breakpoints

- **Definition**: Media breakpoints are specific points in a website's design where the layout and content adjust to accommodate different screen sizes. There are some general breakpoints that you can use to target phones, tablets and desktop screens. But it is not wise to try to chase down every single possible screen size for different devices.

```
/* Styles for screens wider than 768px */
@media screen and (min-width: 768px) {
  body {
    font-size: 1.125rem;
  }
}
```

- **Small Devices (smartphones)**: up to 640px
- **Medium Devices (tablets)**: 641px to 1024px
- **Large Devices (desktops)**: 1025px and larger

Mobile first approach

- **Definition**: The `mobile-first` approach is a design philosophy and development strategy in responsive web design that prioritizes creating websites for mobile devices before designing for larger screens.

```
/* Base styles for mobile */
.container {
  width: 100%;
  padding: 10px;
}

/* Styles for larger screens */
@media screen and (min-width: 768px) {
  .container {
    width: 750px;
    margin: 0 auto;
    padding: 20px;
  }
}

@media screen and (min-width: 1024px) {
  .container {
    width: 960px;
  }
}
```

--assignment--

Review the Responsive Web Design topics and concepts.

Searching and Sorting Algorithms Review

Searching Algorithms

Searching algorithms let you search for a target within a certain list of items.

In computer science, there are two searching algorithms you should know about. They are **linear search** and **binary search** algorithms. It is important to understand the differences between the two algorithms

and when to use each one.

Linear Search

- Linear search iterates through a list of items, checking each item from the beginning until the target item is found.
- If the target item is found, the index where it is located in the list is returned.
- If the target is not found, it returns `-1`, which means **invalid index** in most programming languages.
- Because linear search checks each item until it finds the target, it is not efficient for a large list of items.
- The time complexity of linear search is `O(n)` because the time it takes to search through the list grows linearly with the size of the list.
- The space complexity of linear search is `O(1)` because it doesn't require any additional space to search through the list.

Binary Search

- Binary search works by dividing a list of items in half, and checking if the target value is in the middle of the list.
- The condition for binary search to work is that the items in the list must be in ascending order.
- Binary search is a more efficient algorithm for searching through a large list of items because it divides the list of items in half and ignores any half where the target is not found.
- If the target item is found in the middle of the list, the index of the target item is returned.
- If the item is not found, the algorithm checks if the target item is in the left or right half of the list.
- It continues to divide the remaining parts of the list into halves until the target item is found.
- If the target item is finally not found in the list, it returns `-1`
- The time complexity of binary search is `O(log n)` because the time it takes to search through the list grows logarithmically with the size of the list.
- The space complexity of binary search is `O(1)` because it doesn't require any additional space to search through the list.

How Linear Search Differs from Binary Search

- Binary search is more suitable for a large list of items compared to linear search.
- The time complexity of linear search is `O(n)` because the time it takes to search through the list grows linearly with the size of the list.
- The time complexity of binary search is `O(log n)` because the time it takes to search through the list grows logarithmically with the size of the list.

Sorting Algorithms and Divide-and-Conquer

In computer science, divide-and-conquer is a technique used to break down a problem into smaller sub-problems so they are easier to solve. Recursion is the technique often employed in divide-and-conquer, and divide-and-conquer is a powerful strategy used to implement many efficient sorting algorithms like merge sort.

Merge Sort

- Merge sort is a sorting algorithm that follows the divide-and-conquer approach.
- It works by recursively dividing a list into smaller sub-lists until each sub-list contains only one element.
- It then repeatedly merges the sub-lists back together in a sorted order.
- The time complexity for merge sort is `O(n log n)` because the list is continuously divided in half `(log n)` and then merged together `(O(n))`.
- The space complexity of merge sort is `O(n)` because it is not an in-place sorting algorithm.

--assignment--

Review the Searching and Sorting Algorithms topics and concepts.

Semantic HTML Review

Importance of Semantic HTML

- **Structural hierarchy for heading elements**: It is important to use the correct heading element to maintain the structural hierarchy of the content. The `h1` element is the highest level of heading and the `h6` element is the lowest level of heading.
- **Presentational HTML elements**: Elements that define the appearance of content. Ex. the deprecated `center`, `big` and `font` elements.
- **Semantic HTML elements**: Elements that hold meaning and structure. Ex. `header`, `nav`, `figure`.

Semantic HTML Elements

- **Header element**: used to define the header of a document or section.
- **Main element**: used to contain the main content of the web page.
- **Section element**: used to divide up content into smaller sections.
- **Navigation Section (`nav`) element**: represents a section with navigation links.
- **Figure element**: used to contain illustrations and diagrams.
- **Emphasis (`em`) element**: marks text that has stress emphasis.

```
<p>
  Never give up on <em>your</em> dreams.
</p>
```

- **Idiomatic Text (`i`) element**: used for highlighting alternative voice or mood, idiomatic terms from another language, technical terms, and thoughts.

```
<p>
  There is a certain <i lang="fr">je ne sais quoi</i> in the air.
</p>
```

The `lang` attribute inside the open `i` tag is used to specify the language of the content. In this case, the language would be French. The `i` element does not indicate if the text is important or not, it only shows that it's somehow different from the surrounding text.

- **Strong Importance (`strong`) element**: marks text that has strong importance.

```
<p>
  <strong>Warning:</strong> This product may cause allergic reactions.
</p>
```

- **Bring Attention To (`b`) element**: used to bring attention to text that is not important for the meaning of the content. It's commonly used to highlight keywords in summaries or product names in reviews.

```
<p>
  We tested several products, including the <b>SuperSound 3000</b> for audio
quality, the <b>QuickCharge Pro</b> for fast charging, and the <b>Ecoclean
Vacuum</b> for cleaning. The first two performed well, but the <b>Ecoclean
Vacuum</b> did not meet expectations.
</p>
```

- **Description List (`dl`) element**: used to represent a list of term-description groupings.
- **Description Term (`dt`) element**: used to represent the term being defined.
- **Description Details (`dd`) element**: used to represent the description of the term.

```
<dl>
  <dt>HTML</dt>
  <dd>HyperText Markup Language</dd>
  <dt>CSS</dt>
  <dd>Cascading Style Sheets</dd>
</dl>
```

- **Block Quotation (`blockquote`) element**: used to represent a section that is quoted from another source. This element has a `cite` attribute. The value of the `cite` attribute is the URL of the source.

```
<blockquote cite="https://www.freecodecamp.org/news/learn-to-code-book/">
  "Can you imagine what it would be like to be a successful developer? To
have built software systems that people rely upon?"
</blockquote>
```

- **Citation (`cite`) element**: used to attribute the source of the referenced work visually. Marks up the title of the reference.

```
<div>
  <blockquote cite="https://www.freecodecamp.org/news/learn-to-code-book/">
    "Can you imagine what it would be like to be a successful developer? To
have built software systems that people rely upon?"
  </blockquote>
  <p>
    -Quincy Larson, <cite>How to learn to Code and Get a Developer Job [Full
Book].</cite>
  </p>
</div>
```

- **Inline Quotation (`q`) element**: used to represent a short inline quotation.

```
<p>
  As Quincy Larson said,
  <q cite="https://www.freecodecamp.org/news/learn-to-code-book/">
    Momentum is everything.
  </q>
</p>
```

- **Abbreviation (`abbr`) element**: used to represent an abbreviation or acronym. To help users understand what the abbreviation or acronym is, you can show its full form, a human readable description, with the `title` attribute.

```
<p>
  <abbr title="HyperText Markup Language">HTML</abbr> is the foundation of
the web.
</p>
```

- **Contact Address (`address`) element**: used to represent the contact information.

- **(Date) Time (`time`) element**: used to represent a date and/or time. The `datetime` attribute is used to translate dates and times into a machine-readable format.

```
<p>
  The reservations are for the <time datetime="20:00">20:00 </time>
</p>
```

- **ISO 8601 Date (`datetime`) attribute**: used to represent dates and times in a machine-readable format. The standard format is `YYYY-MM-DDThh:mm:ss`, with the capital `T` being a separator between the date and time.
- **Superscript (`sup`) element**: used to represent superscript text. Common use cases for the `sup` element would include exponents, superior lettering and ordinal numbers.

```
<p>
  2<sup>2</sup> (2 squared) is 4.
</p>
```

- **Subscript (`sub`) element**: used to represent subscript text. Common use cases for the subscript element include chemical formulas, footnotes, and variable subscripts.

```
<p>
  CO<sub>2</sub>
</p>
```

- **Inline Code (`code`) element**: used to represent a fragment of computer code.
- **Preformatted Text (`pre`) element**: represents preformatted text

```
<pre>
  <code>
    body {
      color: red;
    }
  </code>
</pre>
```

- **Unarticulated Annotation (`u`) element**: used to represent a span of inline text which should be rendered in a way that indicates that it has a non-textual annotation.

```
<p>
  You can use the unarticulated annotation element to highlight
  <u>inccccort</u> <u>spling</u> <u>issses</u>.
</p>
```

- **Ruby Annotation (`ruby`) element**: used for annotating text with pronunciation or meaning explanations. An example usage is for East Asian typography.
- **Ruby fallback parenthesis (`rp`) element**: used as a fallback for browsers lacking support for displaying ruby annotations.
- **Ruby text (`rt`) element**: used to indicate text for the ruby annotation. Usually used for pronunciation, or translation details in East Asian typography.

```

```

```
<ruby>
  明日 <rp>(</rp><rt>Ashita</rt><rp>)</rp>
</ruby>
```

- **Strikethrough (`s`) element**: used to represent content that is no longer accurate or relevant.

```
<p>
  <s>Tomorrow's hike will be meeting at noon.</s>
</p>
<p>
  Due to unforeseen weather conditions, the hike has been canceled.
</p>
```

--assignment--

Review the Semantic HTML topics and concepts.

Styling Forms Review

Best Practices for Styling Inputs

- **Styling Inputs**: As with all text elements, you need to ensure the styles you apply to text inputs are accessible. This means the font needs to be adequately sized, and the color needs to have sufficient contrast with the background. Input elements are also focusable. When you are editing your styles, you should take care that you preserve a noticeable indicator when the element has focus, such as a bold border.

Using `appearance: none` for Inputs

- **`appearance: none`**: Browsers apply default styling to a lot of elements. The `appearance: none` CSS property gives you complete control over the styling, but comes with some caveats. When building custom styles for input elements, you will need to make sure focus and error indicators are still present.

Commons Issues Styling `datetime-local` and `color` Properties

- **Common Issues**: These special types of inputs rely on complex pseudo-elements to create things like date and color pickers. This presents a significant challenge for styling these inputs. One challenge is that the default styling is entirely browser-dependent, so the CSS you write to make the picker look the way you intend may be entirely different on another browser.

--assignment--

Review the Styling Forms topics and concepts.

Testing Review

Manual and Automated Testing

- **Manual Testing**: In manual testing, a tester will manually go through each part of the application and test out different features to make sure it works correctly. If any bugs are uncovered in the

testing process, the tester will report those bugs back to the software team so they can be fixed.
- **Automated Testing**: In automated testing, you can automate your tests by writing a separate program that checks whether your application behaves as expected.

Unit Testing

- **Unit Testing**: In unit testing, you test each function to ensure that everything is working as expected. Unit tests can also serve as a form of documentation for your application because they are meant to represent the expected behavior for your code.
- **Single Responsibility Principle**: The single responsibility principle recommends keeping each function small and responsible for one thing.
- **Common JavaScript Testing Frameworks**: Some common testing frameworks include Jest, Mocha, and Vitest. Jest is a popular testing framework for unit tests.

Here is an example of unit tests using Jest.

First, you can create a function that is responsible for returning a newly formatted string:

```
export function getFormattedWord(str) {
  if (!str) return '';
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

In a separate `getFormattedWord.test.js` file, you can write some tests to verify that the function is working as expected. The `getFormattedWord.test.js` file will look like this:

```
import { getFormattedWord } from "./getFormattedWord.js";
import { test, expect } from "jest";

test('capitalizes the first letter of a word', () => {
  expect(getFormattedWord('hello')).toBe('Hello');
});
```

- **`expect` Function**: The `expect` function is used to test a value.
- **Matcher**: Matcher is a function that checks whether the value behaves as expected. In the above example, the matcher is `toBe()`. Jest has a variety of matchers.

To use Jest, you first need to install the `jest` package by using `npm i jest`. You will also need to add a script to your `package.json` file like this:

```
"scripts": {
  "test": "jest"
},
```

Then, you can run the command `npm run test` to run your tests.

Software Development Lifecycle

- **Different Stages of Software Development Lifecycle**:
    - **Planning Stage**: The development team collects requirements for the proposed work from the stakeholders.
    - **Design Stage**: The software team breaks down the requirements and decides on the best approaches for solutions.
    - **Implementation Stage**: The software team breaks down the requirements into manageable tasks and implements them.

    - **Testing Stage**: This involves manual and automated testing for the new work. Sometimes, the team tests out the application throughout the entire development stage to catch and fix any issues that come up.
    - **Deployment Stage**: The team deploys the new changes to a build or testing environment.
    - **Maintenance Stage**: This involves fixing any issues that arise from customers in the production application.

- **Different Models of the Software Development Lifecycle**:

    - **Waterfall Model**: The Waterfall model is where each phase of the lifecycle needs to be completed before the next phase can begin.
    - **Agile Model**: The Agile model focuses on iterative development by breaking down work into sprints.

BDD and TDD

- **TDD**: Test-driven development is a methodology that emphasizes writing tests first. Writing tests before building out features provides real-time feedback to developers during the development process.
- **BDD**: Behavior-driven development is the approach of aligning a series of tests with business goals. The test scenarios in BDD should be written in a language that can be understood by both technical and non-technical individuals. An example of such syntax is Gherkin.
- **BDD Testing Frameworks**: Examples of BDD testing frameworks include Cucumber, JBehave, and SpecFlow.

Assertions in Unit Testing

- **Assertion**: Assertions are used to test that the code is behaving as expected.
- **Assertion Libraries**: Chai is a commonly used assertion library. Other common JavaScript assertion libraries are `should.js` and `expect.js`.

Here is an example of an assertion using Chai that checks that the return value from the `addThreeAndFour` function is equal to the number 7:

```
assert.equal(addThreeAndFour(), 7);
```

- **Best Practices**: Regardless of which assertion library you use, you should write clear assert and failure messages that will help you understand which tests are failing and why.

Mocking, Faking, and Stubbing

- **Mocking**: Mocking is the process of replacing real data with false data that simulates the behavior of real components. For example, you could mock the API response in testing instead of making continuous API calls to fetch the data.
- **Stubbing**: Stubs are objects that return pre-defined responses or dummy data for an expected behavior in an application. For example, you can stub the behavior for a database connection in your tests without having to rely on an actual database connection.
- **Faking**: Fakes are simplified versions of real components without the complexity or side effects. For example, you can fake a database by storing the data in memory instead of interacting with the real database. This will allow you to mimic database operations in memory, which will be much faster than dealing with the real database.

Functional Testing

- **Functional Testing**: Functional testing checks if the features and functions of the application work as expected. The goal of functional testing is to test the system as a whole against multiple scenarios.
- **Non-Functional Testing**: Non-functional testing focuses on things like performance and reliability.

- **Smoke testing**: Smoke testing is a preliminary check on the system for basic or critical issues before beginning more extensive testing.

End-to-End Testing

- **End-to-End Testing**: End-to-end testing, or E2E, tests real-world scenarios from the user's perspective. End-to-end tests help ensure that your application behaves correctly and is predictable for users. However, it is time-consuming to set up, design, and maintain.
- **End-to-End Testing Frameworks**: Playwright is a popular end-to-end testing framework developed by Microsoft. Other examples of end-to-end testing tools include Cypress, Selenium, and Puppeteer.

Here is an example of E2E tests from the freeCodeCamp codebase using Playwright.
The `beforeEach` hook will run before each of the tests. The tests check that the donor has a supporter link in the menu bar, as well as a special stylized border around their avatar:

```
test.beforeEach(async ({ page }) => {
  execSync("node ./tools/scripts/seed/seed-demo-user --set-true isDonating");
  await page.goto("/donate");
});

...

test("The menu should have a supporters link", async ({ page }) => {
  const menuButton = page.getByTestId("header-menu-button");
  const menu = page.getByTestId("header-menu");

  await expect(menuButton).toBeVisible();
  await menuButton.click();

  await expect(menu).toBeVisible();

  await expect(page.getByRole("link", { name: "Supporters" })).toBeVisible();
});

test("The Avatar should have a special border for donors", async ({ page })
=> {
  const container = page.locator(".avatar-container");
  await expect(container).toHaveClass("avatar-container gold-border");
});
```

Usability Testing

- **Usability Testing**: Usability testing is when you have real users interacting with the application to discover if there are any design, user experience, or functionality issues in the app. Usability testing focuses on the intuitiveness of the application by users.
- **Four Common Types of Usability Testing**:
  - **Explorative**: Explorative usability testing involves users interacting with the different features of the application to better understand how they work.
  - **Comparative**: Comparative testing is where you compare your application's user experience with similar applications in the marketplace.
  - **Assessment**: Assessment testing is where you study how intuitive the application is to use.
  - **Validation**: Validation testing is where you identify any major issues that will prevent the user from using the application effectively.
- **Usability Testing Tools**: Examples of tools for usability testing include Loop11, Maze, Userbrain, UserTesting, and UXTweak.

Compatibility Testing

- **Compatibility Testing**: The goal of compatibility testing is to ensure that your application works in different computing environments.
- **Different Types of Compatibility Testing**:
    - **Backwards Compatibility**: Backwards compatibility refers to when the software is compatible with earlier versions.
    - **Forwards Compatibility**: Forwards compatibility refers to when the software and systems will work with future versions.
    - **Hardware Compatibility**: Hardware compatibility is the software's ability to work properly in different hardware configurations.
    - **Operating Systems Compatibility**: Operating systems compatibility is the software's ability to work on different operating systems, such as macOS, Windows, and Linux distributions like Ubuntu and Fedora.
    - **Network Compatibility**: Network compatibility means the software can work in different network conditions, such as different network speeds, protocols, security settings, etc.
    - **Browser Compatibility**: Browser compatibility means the web application can work consistently across different browsers, such as Google Chrome, Safari, Firefox, etc.
    - **Mobile Compatibility**: It is important to ensure that your software applications work on a variety of Android and iOS devices, including phones and tablets.

Performance Testing

- **Performance Testing**: In performance testing, you test an application's speed, responsiveness, scalability, and stability under different workloads. The goal is to resolve any type of performance bottleneck.
- **Different Types of Performance Testing**:
    - **Load Testing**: Load testing determines how a system behaves during normal and peak load times.
    - **Stress Testing**: Stress testing is where you test your application in extreme loads and see how well your system responds to the higher load.
    - **Soak Testing (Endurance Testing)**: Soak testing or endurance testing is a type of load testing where you test the system with a higher load for an extended period of time.
    - **Spike Testing**: Spike testing is where you dramatically increase and decrease the loads and analyze the system's reactions to the changes.
    - **Breakpoint Testing (Capacity Testing)**: Breakpoint testing or capacity testing is where you slowly increment the load over time to the point where the system starts to fail or degrade.

Security Testing

- **Security Testing**: Security testing helps identify vulnerabilities and weaknesses.
- **Security Principles**:
    - **Confidentiality**: This protects against the release of sensitive information to other recipients that aren't the intended recipient.
    - **Integrity**: This involves preventing malicious users from modifying user information.
    - **Authentication**: This involves verifying the user's identity to ensure that they are allowed to use that system.
    - **Authorization**: This is the process of determining what actions authenticated users are allowed to perform or which parts of the system they are permitted to access.
    - **Availability**: This ensures that information and services are available to authorized users when they need it.
    - **Non-Repudiation**: This ensures that both the sender and recipient have proof of delivery and verification of the sender's identity. It protects against the sender denying having sent the information.
- **Common Security Threats**:
    - **Cross-Site Scripting (XSS)**: XSS attacks happen when an attacker injects malicious scripts into a web page and then executes them in the context of the victim's browser.
    - **SQL Injection**: SQL injection allows malicious users to inject malicious code into a database.

    - **Denial-of-Service (DoS) Attack**: DoS attack is when malicious users flood a website with a high number of requests or traffic, causing the server to slow down and possibly crash, making the site unavailable to users.
- **Categories of Security Testing Tools**:
    - **Static Application Security Testing**: These tools evaluate the source code for an application to identify security vulnerabilities.
    - **Dynamic Application Security Testing**: These tools interface with the application's frontend to uncover potential security weaknesses. DAST tools do not have access to the source code.
- **Penetration Testing (pentest)**: Penetration testing is a type of security testing that involves creating simulated cyberattacks on the application to identify any vulnerabilities in the system.

A/B Testing

- **A/B Testing**: A/B testing involves comparing two versions of a page or application and studying which version performs better. It is also known as bucket or split testing. A/B testing allows you to make more data-driven decisions and continually improve the user experience.
- **Tools for A/B Testing**: Examples of tools to use for A/B testing include GrowthBook and LaunchDarkly.

Alpha and Beta Testing

Once the initial development and software testing are complete, it is important to have the application tested by testers and real users. This is where alpha and beta testing come in.

- **Alpha Testing**: Alpha testing is done by a select group of testers who go through the application to ensure there are no bugs before it is released into the marketplace. Alpha testing is part of acceptance testing and utilizes both white and black box testing techniques.
- **Beta Testing**: Beta testing is where the application is made available to real users. Users can interact with the application and provide feedback. Beta testing is also a form of user acceptance testing.
- **Acceptance Testing**: Acceptance testing ensures that the software application meets the business requirements and the needs of users before its release.
- **Black Box Testing**: Black box testing only focuses on the expected behavior of the application.
- **White Box Testing**: White box testing involves the tester knowing the internal components and performing tests on them.

Regression Testing

- **Regression**: Regression refers to situations where new changes unintentionally break existing functionality.
- **Regression Testing**: Regression testing helps catch regression issues. In regression testing, you re-run functional tests against parts of your application to ensure that everything still works as expected.
- **Tools for Regression Testing**: Tools that you can use to perform regression testing include Puppeteer, Playwright, Selenium, and Cypress.
- **Techniques for Regression Testing**:
    - **Unit Regression Testing**: This is where you have a list of items that need to be tested each time major changes or fixes are implemented into the app.
    - **Partial Regression Testing**: This involves targeted approaches to ensure that new changes have not broken specific aspects of the application.
    - **Complete Regression Testing**: This runs tests against all the functionalities in the codebase. This is the most time-consuming and detailed option.
- **Retesting**: Retesting is used to check for known issues and ensure that they have been resolved. In contrast, regression testing searches for unknown issues that might have occurred through recent changes in the codebase.

--assignment--

Review the Testing topics and concepts.

Typescript Review

What is TypeScript

- **JavaScript**: JavaScript is a dynamically-typed language. This means that variables can receive any values at run time. The challenge of a dynamically-typed language is that the lack of type safety can introduce errors.

For example, even if your JavaScript function expects an array, you can still call it with a number:

```
const getRandomValue = (array) => {
  return array[Math.floor(Math.random() * array.length)];
}

console.log(getRandomValue(10));
```

The `console` output for the example above will be `undefined`.

- **TypeScript**: TypeScript extends the JavaScript language to include static typing, which helps catch errors caused by type mismatches before you run your code.

For example, you can define a type for the `array` parameter as follows:

```
const getRandomValue = (array: string[]) => {
  return array[Math.floor(Math.random() * array.length)];
}
```

This type definition tells TypeScript that the `array` parameter must be an array of strings. Then, when you call `getRandomValue` and pass it a number, you get an error called a compiler error.

- **Compiler**: You first need to compile TypeScript code into regular JavaScript. When you run the compiler, TypeScript will evaluate your code and throw an error for any issues where the types don't match.

Data Types in TypeScript

- **Primitive Data Types in TypeScript**: For the primitive data types `string`, `null`, `undefined`, `number`, `boolean`, and `bigint`, TypeScript offers corresponding type keywords.

```
let str: string = "Naomi";
let num: number = 42;
let bool: boolean = true;
let nope: null = null;
let nada: undefined = undefined;
```

- **Array**: You can define an array of specific type with two different syntaxes.

```
const arrOne: string[] = ["Naomi"];
const arrTwo: Array<string> = ["Camperchan"];
```

- **Objects**: You can define the exact structure of an object.

```
const obj: { a: string, b: number } = { a: "Naomi", b: 42 };
```

If you want an object with any keys, but where all values must be strings, there are two ways to define it:

```
const objOne: Record<string, string> = {};
const objTwo: { [key: string]: string } = {};
```

- **Other Helpful Types in TypeScript**:
  - **`any`**: `any` indicates that a value can have any type. It tells the compiler to stop caring about the type of that variable.
  - **`unknown`**: `unknown` tells TypeScript that you *do* care about the type of the value, but you don't actually know what it is. `unknown` is generally preferred over `any`.
  - **`void`**: This is a special type that you'll typically only use when defining functions. Functions that don't have a return value use a return type of `void`.
  - **`never`**: It represents a type that will never exist.

- **`type` Keyword**: This keyword is like `const`, but instead of declaring a variable, you can declare a type.

It is useful for declaring custom types such as union types or types that include only specific values:

```
type stringOrNumber = string | number;
type bot = "camperchan" | "camperbot" | "naomi";
```

- **`interface`**: Interfaces are like classes for types. They can implement or extend other interfaces, are specifically object types, and are generally preferred unless you need a specific feature offered by a `type` declaration.

```
interface wowie {
  zowie: boolean;
  method: () => void;
}
```

- **Defining Return Type**: You can also define the *return type* of the function.

The example below defines the return value as a string. If you try to return anything else, TypeScript will give a compiler error.

```
const getRandomValue = (array: string[]): string => {
  return array[Math.floor(Math.random() * array.length)];
}
```

Generics

- **Defining Generic Type**: You can define a generic type and refer to it in your function. It can be thought of as a special parameter you provide to a function to control the behavior of the function's type definition.

Here is an example of defining a generic type for a function:

```
const getRandomValue = <T>(array: T[]): T => {
  return array[Math.floor(Math.random() * array.length)];
}
const val = getRandomValue([1, 2, 4])
```

The `<T>` syntax tells TypeScript that you are defining a generic type `T` for the function. `T` is a common name for generic types, but you can use any name.

Then, you tell TypeScript that the `array` parameter is an array of values matching the generic type, and that the returned value is a single element of that same type.

- **Specifying Type Argument in Function Call**: You can pass a type argument to a function call using angle brackets between the function name and its parameters.

Here is an example of passing a type argument to a function call:

```
const email = document.querySelector<HTMLInputElement>("#email");
console.log(email.value);
```

This tells TypeScript that the element you expect to find will be an input element.

Type Narrowing

- **Narrowing by Truthiness**: In the example below, you get a compiler error trying to access the `value` property of `email` because `email` *might* be `null`.

```
const email = document.querySelector<HTMLInputElement>("#email");
console.log(email.value);
```

You can use a conditional statement to confirm `email` is *truthy* before accessing the property:

```
const email = document.querySelector<HTMLInputElement>("#email");
if (email) {
  console.log(email.value);
}
```

Truthiness checks can also work in the reverse direction:

```
const email = document.querySelector<HTMLInputElement>("#email");
if (!email) {
  throw new ReferenceError("Could not find email element!")
}
console.log(email.value);
```

Throwing an error ends the logical execution of this code, which means when you reach the `console.log()` call, TypeScript knows `email` *cannot* be `null`.

- **Optional Chaining**: Optional chaining `?.` is also a form of type narrowing, under the same premise that the property access can't happen if the `email` value is `null`.

```
const email = document.querySelector<HTMLInputElement>("#email");
console.log(email?.value);
```

- **`typeof` Operator**: You can use a conditional to check the type of the variable using the `typeof` operator.

```
const myVal = Math.random() > 0.5 ? 222 : "222";
if (typeof myVal === "number") {
  console.log(myVal / 10);
}
```

- **`instanceof` Keyword**: If the object comes from a class, you can use the `instanceof` keyword to narrow the type.

```
const email = document.querySelector("#email");

if (email instanceof HTMLInputElement) {
    console.log(email.value);
}
```

- **Casting**: When TypeScript cannot automatically determine the type of a value, such as the result from `request.json()` method in the example below, you'll get a compiler error. One way to resolve this is by casting the type, but doing so weakens TypeScript's ability to catch potential errors.

```
interface User {
    name: string;
    age: number;
}

const printAge = (user: User) =>
  console.log(`${user.name} is ${user.age} years old!`)

const request = await fetch("url")
const myUser = await request.json() as User;
printAge(myUser);
```

- **Type Guard**: Instead of casting the type, you can write a type guard:

```
interface User {
    name: string;
    age: number;
}

const isValidUser = (user: unknown): user is User => {
  return !!user &&
    typeof user === "object" &&
    "name" in user &&
    "age" in user;
}
```

The `user is User` syntax indicates that your function returns a boolean value, which when true means the `user` value satisfies the `User` interface.

`tsconfig` File

- **`tsconfig.json`**: TypeScript's compiler settings live in a `tsconfig.json` file in the root directory of your project.

```
{
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./prod",
    "lib": ["ES2023"],
    "target": "ES2023",
    "module": "ES2022",
    "moduleResolution": "Node",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "strict": true
  },
  "exclude": ["test/"]
}
```

Here are descriptions of the compiler options used in the example above:

- **`compilerOptions`**: The `compilerOptions` property is where you control how the TypeScript compiler behaves.
- **`rootDir` and `outDir`**: The `rootDir` and `outDir` tell TypeScript which directory holds your source files, and which directory should contain the transpiled JavaScript code.
- **`lib`**: The `lib` property determines which type definitions the compiler uses, and allows you to include support for specific ES releases, the DOM, and more.
- **`module` and `moduleResolution`**: The `module` and `moduleResolution` work in tandem to manage how your package uses modules - either CommonJS or ECMAScript.
- **`esModuleInterop`**: The `esModuleInterop` allows for smoother interoperability between CommonJS and ES modules by automatically creating namespace objects for imports.
- **`skipLibCheck`**: The `skipLibCheck` option skips validating `.d.ts` files that aren't referenced by imports in your code.
- **`strict`**: The `strict` flag enables several checks, such as ensuring proper handling of nullable types and warning when TypeScript falls back to `any`.
- **`exclude`**: The top-level `exclude` property tells the compiler to ignore these TypeScript files during compilation.

--assignment--

Review the Typescript topics and concepts.

Web Performance Review

Differences Between Real and Perceived Performance

- **Perceived Performance**: This is how users perceive the performance of a website. It's how they evaluate it in terms of responsiveness and reliability. This is a subjective measurement, so it's hard to quantify it, but it's very important, since the user experience determines the success or failure of a website.
- **Real Performance**: This is the objective and measurable performance of the website. It's measured using metrics like page load time, server response time, and rendering time. These measurements are influenced by multiple factors related to the network and to the code itself.

Techniques for Improving Perceived Performance

- **Lazy Loading**: This technique reduces the initial load time as much as possible by loading non-essential resources in the background.
- **Minimize Font Delays**: If your website has custom fonts, you should also try to minimize font loading delays, since this may result in flickering or in showing the fallback font while the custom

font is being loaded. A suggestion for this is using a fallback font that is similar to the custom font, so in case this happens, the change will be more subtle.
- **Use of Loading Indicators**: Showing a loading indicator for a long-running process as soon as the user clicks on an element can help the user feel connected and engaged with the process, making the wait time feel shorter.

Core Performance Concepts

- **Source order**: This refers to the way HTML elements are structured in the document. This determines what loads first and can significantly impact performance and accessibility.

Some best practices for source order include:

- Placing critical content such as headings, navigation or main text higher in the HTML structure.
- Deferring non-essential scripts such as ones for analytics, or third-party widgets, so they don't block rendering.
- Using progressive enhancement, to ensure the core experience works even before styles and scripts load. Progressive enhancement is a way of building websites and applications based on the idea that you should make your page work with HTML first.

Here is an example of good source order, using the best practices we just went through:

```
<h1>Welcome to FastSite!</h1>
<p>Critical information loads first.</p>
<script src="slow-script.js" defer></script>
```

- **Critical Rendering Path**: This is the sequence of steps the browser follows to convert code into pixels on the screen.
- **Latency**: This is the time it takes for a request to travel between the browser and the server. So in other words, high latency equals slow pages.

Some ways of reducing latency include:

- Using CDNs, or Content Delivery Networks, to serve files from closer locations.
- Enabling compression using things such as Gzip to reduce file sizes.
- Optimizing images and using lazy loading.

```
<img src="placeholder.jpg" data-src="real-image.jpg" loading="lazy">
```

Improving INP

- **Definition**: INP (Interaction to Next Paint) assesses a page's overall responsiveness by measuring the time from when a user interacts, like a click or key press, to the next time the browser updates the display. A lower INP indicates a more responsive page.

Here are some ways to improve INP:

- Reduce main thread work by breaking up long JavaScript tasks.
- Use `requestIdleCallback()` for non-critical scripts. This will queue a function to be called during a browser's idle periods.
- Defer or lazy-load heavy assets which were covered earlier.
- Optimize event handlers. If these handlers run too frequently or perform heavy operations, they can slow down the page and increase INP. The solution for this is debouncing. Debouncing ensures that the function only runs after the user stops typing for a short delay - so for example 300ms. This prevents unnecessary calculations and improves performance.

How Rendering Works in the Browser

- **How Rendering works**: First the browser parses the HTML and builds the DOM. Next, the browser processes the CSS, constructing the CSS Object Model, or CSSOM. This is another tree structure that dictates how elements should be styled. Finally, the browser paints the pixels to the screen, rendering each element based on the calculated styles and layout. In complex pages, this might involve multiple layers that are composited together to form the final visual output.

How Performance Impacts Sustainability

- **Background Information**: The internet accounts for around 2% of global carbon emissions—that's the same as the airline industry! Every byte transferred requires electricity, from data centers to user devices. Larger files and inefficient scripts mean more power consumption. A high-performance website isn't just faster, it also reduces unnecessary processing and energy use.

Ways to Reduce Page Loading Times

- **Optimize Media Assets**: Large images and videos are common culprits for slow load times. By optimizing these assets, you can significantly speed up your site. This includes things like compressing images, using modern formats like WebP and using lazy loading for assets.
- **Leverage Browser Caching**: Caching allows browsers to store parts of your website locally, reducing load times for returning visitors.
- **Minify and Compress Files**: Reducing the size of your files can lead to quicker downloads. This includes reducing the size of transmitted files and minifying CSS and JavaScript files.

Improving "time to usable"

- **Definition**: "time to usable" is the interval from when a user requests a page to when they can meaningfully interact with it. To improve "time to usable" you can lazy load your asset or minimize render-blocking resources.

Key Metrics for Measuring Performance

- **First Contentful Paint or FCP**: It measures how quickly the first piece of content—text or image—appears on the screen. A good FCP is regarded as a time below 1.8 seconds, and a poor FCP is above 3 seconds. You can check your FCP using Chrome DevTools and checking the performance tab.
- **Total Blocking Time**: This shows how long the main thread is blocked by heavy JavaScript tasks. If Total Blocking Time (TBT) is high, users experience sluggish interactions. To improve TBT, break up long tasks and defer non-essential scripts.
- **Bounce Rate**: This is the percentage of visitors who leave without interacting. If your site has high bounce rates it might be because your page is too slow.
- **Unique Users**: This metric tracks how many individual visitors come to your site. To view the Bounce Rate and Unique Users, you can use Google Analytics. It will allow you to monitor these metrics and improve engagement.

Common Performance Measurement Tools

- **Chrome DevTools**: Chrome DevTools is a built-in tool inside Google Chrome that lets you analyze and debug performance in real-time. DevTools will show loading times, CPU usage, and render delays. It's especially useful for measuring First Contentful Paint, or FCP, is how fast a user sees the first visible content. If your website feels slow, DevTools will help you spot the bottlenecks.
- **Lighthouse**: This is an automated tool that checks performance, SEO, and accessibility.
- **WebPageTest**: This tool lets you test how your site loads from different locations and devices. This tool gives you a detailed breakdown of your site's Speed Index, Total Blocking Time, and other key performance metrics. If you want to know how real users experience your site globally, WebPageTest is the tool for that.
- **PageSpeed Insights**: This tool analyzes your website and suggests quick improvements for both mobile and desktop. It will tell you what's slowing your site down and give specific recommendations like optimizing images, removing render-blocking scripts, and reducing server response times. PageSpeed Insights is a fast and easy way to check how Google sees your site's performance.

- **Real User Monitoring**: RUM tools track actual user behavior, showing how real visitors experience your site. Popular RUM tools include Google Analytics, which tracks page load times and bounce rates, and New Relic or Datadog, which monitor real-time performance issues. If you want data from actual users, RUM tools are essential.

Working with Performance Web APIs

- **Definition**: Performance Web APIs let developers track how efficiently a webpage loads and responds directly in the code. These APIs allow you to measure page load times, track rendering and interaction delays and analyze JavaScript execution time.
- `performance.now()`: This API gives you high-precision timestamps (in milliseconds) to measure how long different parts of your site take to load.

```
const start = performance.now();
// Run some code here
const end = performance.now();

console.log(`Execution time: ${end - start}ms`);
```

- **Performance Timing API**: This API gives you a breakdown of every stage of page loading, from DNS lookup to `DOMContentLoaded`.

```
const timing = performance.timing;

const pageLoadTime = timing.loadEventEnd - timing.navigationStart;
console.log(`Page load time: ${pageLoadTime}ms`);
```

- `PerformanceObserver`: This API listens for performance events such as layout shifts, long tasks, and user interactions.

```
const observer = new PerformanceObserver((list) => {
  list.getEntries().forEach((entry) => {
    console.log(`Long task detected: ${entry.duration}ms`);
  });
});

observer.observe({ type: "longtask", buffered: true });
```

Techniques for Improving CSS Performance

- **CSS Animations**: Animating certain CSS properties, such as dimensions, position, and layout, triggers a process called "reflow", during which the browser recalculates the position and geometry of certain elements on the page. This requires a repaint, which is computationally expensive. Therefore, it's recommended to reduce the number of CSS animations as much as possible or at least give the user an option to toggle them on or off.

Techniques for Improving JavaScript Performance

- **Code Splitting**: Splitting your JavaScript code into modules that perform critical and non-critical tasks is also helpful. This way, you'll be able to preload the critical ones as soon as possible and defer the non-critical ones to render the page as fast as possible.
- **DOM Manipulation**: Remember that DOM Manipulation refers to the process of dynamically changing the content of a page with JavaScript by interacting with the Document Object Model (DOM). Manipulating the DOM is computationally expensive. By reducing the amount of DOM manipulation in your JavaScript code will improve performance.

--assignment--

Review the Web Performance topics and concepts.