Basic CLI Commands For windows

1. **dir**
   List the contents of a directory.
   Syntax: dir [path]

2. **ls**
   Show the list of folders and files. (Requires Git Bash or similar tools on Windows)
   Syntax: ls [options] [path]

3. **cd**
   Change the current directory.
   Syntax: cd [directory]

4. **mkdir or md**
   Create a new directory.
   Syntax: mkdir [directory] or md [directory]

5. **rmdir or rd**
   Remove a directory.
   Syntax: rmdir [directory] or rd [directory]

6. **del or erase**
   Delete one or more files.
   Syntax: del [file] or erase [file]

7. **ren or rename**
   Rename a file or directory.
   Syntax: ren [old_name] [new_name] or rename [old_name] [new_name]

8. **type**
   Display the contents of a text file.
   Syntax: type [file]

9. **cls**
   Clear the screen.
   Syntax: cls

10. **exit**
    Exit the command
    prompt.
    Syntax: exit

React js based Terminal commands

1. **node -v**
   To check node version.

2. **npm -v**
   To check Node package manager Version in node terminal / cmd

3. **npm install -g npm@latest**
   In case of npm error …. One time command after node installation.

4. **npx create-react-app [FolderName/Path]**
   Command for create and execute new react-app

5. **npm install -g package-name**
   To install a package globally on the local system.

6. **npm start**
   To run react app on browser.

7. **ctrl + c**
   To Stop react app on browser.

8. **npm install package-name**
   To install npm packages (dependencies) on local folder/application.

9. **npm update**
   To update all npm packages in project to the latest version.

10. **npm run build**
    To deploy the application, build it for the production

Advanced JavaScript Notes

1. **Describe about Map, Filter, Find and Reduce methods**
   These array methods provide powerful tools for working with collections.

   - **Map**
     The map() method of Array creates a new array populated with the results of calling a provided function on every element in the calling array.

```
javascript
const numbers = [1, 2, 3];
const doubled = numbers.map( (n) => {
   return n * 2
});
console.log(doubled);  // [2, 4, 6]
```

   - **Filter**
     The filter() method of Array instances creates a shallow copy of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function.

```
javascript
const numbers = [1, 2, 3, 4];
const even = numbers.filter((n) => n % 2 === 0);
console.log(even);  // [2, 4]
```

   - **ForEach**
     This method is typically used to perform actions or side effects for each element in the array. Unlike methods such as map or filter, forEach does not return a new array; it simply iterates over the array and applies the provided function to each element.

```
const fruits = ['apple', 'banana', 'cherry'];

fruits.forEach((fruit, index) => {

    console.log(`Fruit at index ${index} is ${fruit}`);
});
// Output:
// Fruit at index 0 is apple
// Fruit at index 1 is banana
// Fruit at index 2 is cherry
```

- **Reduce**
  The reduce() method of Array instances executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

```
javascript
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, n) => acc + n, 0);
console.log(sum);  // 10
```

- **Find**
  The find method returns the value of the first element in the array that satisfies the provided testing function. If no values satisfy the testing function, undefined is returned.

```
Example:
javascript

const numbers = [1, 2, 3, 4, 5];

const firstEvenNumber = numbers.find((num) => {return num % 2 === 0});

console.log(firstEvenNumber); // 2
```

2. Spread and Rest Operators

- **Spread Operator**
  The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

```
javascript
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
console.log(arr2);  // [1, 2, 3, 4]
```

- **Rest Operator**
  The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

```
javascript
function sum(...numbers) {
    return numbers.reduce((acc, number) => acc + number, 0);
}

console.log(sum(1, 2, 3));  // 6
```

Looping Objects: Object Keys, Values, and Entries

JavaScript provides several methods to loop over objects' properties:

1. Object.keys(): Returns an array of the object's own enumerable property names.

```
javascript
const obj = { a: 1, b: 2, c: 3 };
Object.keys(obj).forEach(key => {
  console.log(key); // 'a', 'b', 'c'
});
```

2. Object.values(): Returns an array of the object's own enumerable property values.

```
javascript
Object.values(obj).forEach(value => {
  console.log(value); // 1, 2, 3
});
```

3. Object.entries(): Returns an array of the object's own enumerable property [key, value] pairs.

```
javascript
Object.entries(obj).forEach(([key, value]) => {
  console.log(key, value); // 'a' 1, 'b' 2, 'c' 3
});
```

3. **Export Modules**
   Modules allow for encapsulation of code, making it easier to manage and reuse.

```
javascript
// module.js
module.exports = function() {
    console.log('Hello from a module!');
};

// app.js
const myModule = require('./module.js');
myModule();
```

4. **Destructuring**
   Destructuring allows for the unpacking of values from arrays or properties from objects into distinct variables.

   - **Array Destructuring**

```
javascript
const [a, b] = [1, 2];
console.log(a, b);  // 1 2
```

   - **Object Destructuring**

```
javascript
const {name, age} = {name: 'John', age: 30};
console.log(name, age);  // John 30
```

5. **try-Catch Structure in js**
   The try statement allows you to define a block of code to be tested for errors while it is being executed.
   The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

```
try {
  // Code that may throw an error
  let result = riskyOperation();
  console.log("Operation successful, result:", result);
} catch (error) {
  // Code to handle the error
  console.error("An error occurred:", error.message);
} finally {
  // Code that will run regardless of the try / catch result
  console.log("Cleanup code, if any.");
}

function riskyOperation() {
  // Simulating an error
  throw new Error("Something went wrong!");
}
```

6. **Asynchronous JavaScript**
   Asynchronous JavaScript involves operations that are executed after a certain period, allowing for non-blocking code execution.

   - **i. By using Callback Function**

   A callback is a function passed as an argument to another function, to be executed after the first function completes.

```
javascript
function doSomething(callback) {
    console.log('Doing something...');
    callback();
}

function notify() {
    console.log('Done!');
}

doSomething(notify);
```

   - **ii. By using Promises**

   A promise represents an operation that hasn't completed yet but is expected in the future.

```
javascript
const promise = new Promise((resolve, reject) => {
    const success = true;
    if (success) {
        resolve('Operation was successful.');
    } else {
        reject('Operation failed.');
    }
});

promise
    .then((message) => console.log(message))
    .catch((error) => console.error(error));
```

   - **iii. By using Async/Await**

   Async/Await is syntactic sugar over promises, making asynchronous code look synchronous.

```
javascript
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 fetching data:', error);
    }
}

fetchData();


// module.js
export function greet() {
    console.log('Hello from a module!');
}

// app.js
import { greet } from './module.js';
greet();
```

7. **Short Circuiting (&& and ||) in JavaScript**

Short-circuit evaluation is a feature of logical operators in JavaScript that stops the evaluation as soon as the outcome is determined.

- **- AND (`&&`) operator:** Returns the first falsy value or the last value if none are falsy.

```
  javascript
  let a = false && true; // a is false
  let b = true && false; // b is false
  let c = true && true;  // c is true
  let d = false && 'hello'; // d is false
  let e = 'hello' && 'world'; // e is 'world'
```

- **- OR (`||`) operator:** Returns the first truthy value or the last value if none are truthy.

```
  javascript
  let a = false || true; // a is true
  let b = false || false; // b is false
```

```
  let c = true || false; // c is true
  let d = '' || 'hello'; // d is 'hello'
  let e = 'hello' || 'world'; // e is 'hello'
```

8. **Optional Chaining (`?.`)**

   Optional chaining is available in JavaScript and is used to safely access deeply nested properties of an object without having to explicitly check for the existence of each property.

```
- Syntax:
  javascript

  let user = {
    profile: {
      address: {
        city: 'New York'
      }
    }
  };
  let city = user?.profile?.address?.city; // 'New York'
  let zip = user?.profile?.address?.zip; // undefined
  let street = user?.profile?.street?.name; // undefined
```

9. **Closures**
   A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope. This allows for private variables and encapsulation.

```
javascript
function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
        console.log('Outer Variable: ' + outerVariable);
        console.log('Inner Variable: ' + innerVariable);
    }
}

const newFunction = outerFunction('outside');
newFunction('inside');
// Outer Variable: outside
// Inner Variable: inside
```

10. **Template Literals / String Literals**

    By using backtick we can concatinate with two string

    Template literals allow for multi-line strings and embedded expressions.

```
const website = 'Ticer'
const message = `Welcome to ${website}`
Const

console.log(message)
```

11. Statements Vs. Expressions

1. Statement

A statement is an instruction that performs an action. Statements form the building blocks of a program and are executed to make things happen. They do not return values and are primarily used to control the flow of the program.

```
Example of statement
let x = 10;
```

2. Expression

An expression is any valid unit of code that resolves to a value. Expressions can be used wherever values are expected. They produce values and can be nested inside other expressions or statements.

```
Example of Expression
Const result = 5 + 10
```

12. Turnary Operator

In JavaScript, a ternary operator can be used to replace certain types of if..else statements. For example,

```
// check the age to determine the eligibility to vote
let age = 15;
let result;

if (age >= 18) {
      result = "You are eligible to vote.";
} else {
      result = "You are not eligible to vote yet.";
}

console.log(result);
```

With

```
// ternary operator to check the eligibility to vote
let age = 15;
let result =
    (age >= 18) ? "You are eligible to vote." : "You are not eligible to vote
yet";
console.log(result);
```

13. Prototypes and Inheritance in JS

JavaScript objects inherit properties and methods from a prototype. Prototypal inheritance allows for the sharing of methods and properties across instances.

```
javascript
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.greet = function() {
    console.log('Hello, my name is ' + this.name);
};

const john = new Person('John', 30);
john.greet();  // Hello, my name is John
```

14. Event Loop

The event loop is what allows JavaScript to perform non-blocking operations by handling asynchronous callbacks.

- **Call Stack:** Where the execution context is stacked and managed.
- **Web APIs:** Browser-provided APIs that handle asynchronous operations.
- **Callback Queue:** Queue of callback functions to be executed.

Working with APIs in JavaScript

JavaScript can interact with APIs using the `fetch` method or libraries like Axios.

1. **Using `fetch`:**

```
 javascript
   fetch('https://api.example.com/data')
     .then(response => {
       if (!response.ok) {
         throw new Error('Network response was not ok');
       }
       return response.json();
     })
     .then(data => {
       console.log(data);
     })
     .catch(error => {
       console.error('There has been a problem with your fetch operation:',
error);
     });
```

2. **Using Axios library:**

```
  javascript
   // First, include Axios via CDN or npm
   axios.get('https://api.example.com/data')
     .then(response => {
       console.log(response.data);
     })
     .catch(error => {
       console.error('There has been a problem with your Axios request:',
error);
     });
```

OOP Basics in JavaScript

JavaScript supports Object-Oriented Programming (OOP) principles through the use of objects and classes.

1. **Classes:**

```
javascript
 class Animal {
   constructor(name) {
     this.name = name;
   }

   speak() {
     console.log(`${this.name} makes a noise.`);
   }
 }

 let dog = new Animal('Dog');
 dog.speak(); // Dog makes a noise.
```

2. **Inheritance:**
   Inheritance is a programming concept where a new class (the subclass or derived class) automatically inherits the properties and behavior of an existing class (the superclass or base class), allowing for code reuse and a more hierarchical organization of code.

```
javascript
 class Dog extends Animal {
   speak() {
     console.log(`${this.name} barks.`);
   }
 }

 let dog = new Dog('Dog');
 dog.speak(); // Dog barks.
```

3. **Encapsulation (using private fields):**
   Encapsulation is a programming concept where data and its associated methods are bundled into a single unit, protecting the data from external interference and misuse.

```
javascript
  class Person {
    #age;

    constructor(name, age) {
      this.name = name;
      this.#age = age;
    }

    getAge() {
      return this.#age;
    }
  }

  let person = new Person('John', 30);
  console.log(person.getAge()); // 30
```

4. Polymorphism in JavaScript

Polymorphism is a principle in OOP that allows objects of different classes to be treated as objects of a common superclass. It is most commonly achieved through method overriding and interface implementation.

- **Method Overriding**

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

1. Example of Method Overriding:

```
javascript
  class Animal {
    speak() {
      console.log("Animal makes a noise.");
    }
  }

  class Dog extends Animal {
    speak() {
      console.log("Dog barks.");
    }
  }
```

```
class Cat extends Animal {
  speak() {
    console.log("Cat meows.");
  }
}

let animals = [new Animal(), new Dog(), new Cat()];

animals.forEach(animal => animal.speak());
// Output:
// Animal makes a noise.
// Dog barks.
// Cat meows.
```

Html to JSX
https://transform.tools/html-to-jsx

Github link
https://github.com/masifmirza927/first-react

Turnry Operator
https://www.programiz.com/javascript/ternary-operator

Statement vs Expression
https://www.joshwcomeau.com/javascript/statements-vs-expressions/

Template Literals
https://www.freecodecamp.org/news/template-literals-in-javascript/

Object Destructuring
https://www.educative.io/answers/what-is-object-destructuring-in-javascript

React js Documentation
https://react.dev/learn

Github Portfolio link
https://github.com/masifmirza927/portfolio

Credo link
https://themewagon.github.io/credo/