**JavaScript Viva Prep Material:**

**Function:**

1. What is a function in JavaScript?

   A function is a block of code that performs a specific task, and can be reused multiple times throughout your code.

2. What are the benefits of using functions in JavaScript code?

   Functions allow you to write more modular code, making it easier to read, test, and maintain. They also help to reduce code duplication and improve code organization.

3. How do you declare a function in JavaScript?

   Functions in JavaScript can be declared using the "function" keyword, followed by the name of the function, any parameters it accepts, and the code to be executed within the function's curly braces.

```
function functionName(parameters) {
  // code to be executed
}
```

4. How do you call a function in JavaScript?

   Functions are called by using their name, followed by parentheses containing any arguments to be passed to the function.

```
functionName(arguments);
```

5. What is a function parameter and how is it used in JavaScript?

   A function parameter is a variable that is passed to a function when it is called, and can be used within the function to modify its behavior or perform a specific task.

```
function myFunction(parameter1, parameter2) {
  // code to be executed
}
```

6. What is a return statement in JavaScript and how does it work with functions?

   A return statement is used within a function to specify the value that should be returned when the function is called. It can be used to pass data back to the calling code or to exit a function early.

```
function addNumbers(num1, num2) {
  return num1 + num2;
}

var result = addNumbers(5, 10);
console.log(result); // Output: 15
```

7. What is a callback function in JavaScript?

   A callback function is a function that is passed as an argument to another function and is executed when that function is called. Callback functions are commonly used in asynchronous programming and event handling.

```
function myFunction(callback) {
  // code to be executed
  callback();
}

myFunction(function() {
  console.log('Callback function called.');
});
```

8. How do you pass a function as an argument to another function in JavaScript?

   Functions can be passed as arguments to other functions by simply including the function name as an argument when calling the function.

```
function myFunction(callback) {
  // code to be executed
  callback();
}

function myCallback() {
  console.log('Callback function called.');
}

myFunction(myCallback);
```

9. What is a higher-order function in JavaScript?

   A higher-order function is a function that takes one or more functions as arguments or returns a function as its result. Higher-order functions are used extensively in functional programming.

10. What is the difference between a named and an anonymous function in JavaScript?

    A named function is a function that has a specified name, while an anonymous function is a function that does not have a name. Anonymous functions are commonly used as callback functions.

11. What is the scope of a function in JavaScript?

    The scope of a function refers to the set of variables that are visible and accessible within the function. Variables declared within a function are local to that function and cannot be accessed outside of it.

12. How do you access variables outside of a function in JavaScript?

    Variables declared outside of a function have global scope and can be accessed from within any function in the code.

13. How do you create a closure in JavaScript?

    Closures are created by nesting a function within another function and returning the inner function. The inner function retains access to the variables and parameters of the outer function, even after the outer function has completed execution.

14. What is the purpose of the "this" keyword in a function in JavaScript?

    The "this" keyword is used within a function to refer to the object that the function is a method of. It allows you to access and modify object properties within the function.

15. How do you use the "apply" and "call" methods in JavaScript functions?

    The "apply" and "call" methods are used to set the value of "this" within a function and to pass arguments to the function. The difference between the two methods is that "call" takes the arguments individually, while "apply" takes them as an array.

**Function:**

1. What are Arrow functions in JavaScript?

   Arrow functions are a new syntax introduced in ES6 for creating functions in JavaScript. They provide a more concise syntax compared to regular functions and also have some differences in behavior.

2. How are Arrow functions different from regular functions?

   Arrow functions are different from regular functions in a few ways:

- They are more concise and have a shorter syntax.
- They don't have their own "this" keyword, so the value of "this" inside an Arrow function is determined by the surrounding context.
- They don't have an "arguments" object, so you can't use the "arguments" keyword inside an Arrow function.

3. How do you declare an Arrow function in JavaScript?

   An Arrow function can be declared using the following syntax:

```
const functionName = (parameters) => {
  // code to be executed
};
```

4. How do you call an Arrow function in JavaScript?

   An Arrow function can be called in the same way as a regular function:

```
functionName(arguments);
```

5. What is the difference between implicit and explicit returns in Arrow functions?

   An implicit return in an Arrow function is when the value is returned without using the "return" keyword. Here's an example:

```
const addNumbers = (num1, num2) => num1 + num2;
```

   An explicit return is when the "return" keyword is used to return a value. Here's an example:

```
const addNumbers = (num1, num2) => {
  return num1 + num2;
};
```

6. What is the "this" keyword in JavaScript and how is it used in Arrow functions?

   The "this" keyword in JavaScript refers to the object that the function is a method of. In Arrow functions, the value of "this" is determined by the surrounding context, rather than having its own value.

7. Can you use the "arguments" object in Arrow functions?

No, Arrow functions do not have their own "arguments" object, so you cannot use the "arguments" keyword inside an Arrow function.

8. How do you use Arrow functions in conjunction with Array methods like map(), filter(), and reduce()?

   Arrow functions are often used with Array methods like map(), filter(), and reduce() for a more concise syntax. Here's an example using the map() method:

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

const doubledNumbers = numbers.map(num => num * 2);
```

9. Can you use Arrow functions as methods in objects?

   Yes, Arrow functions can be used as methods in objects. Here's an example:

```
const myObject = {
  myFunction: () => {
    console.log('Arrow function used as a method.');
  }
};
```

10. What are some best practices when using Arrow functions in JavaScript?
    Some best practices when using Arrow functions in JavaScript include:

- Using Arrow functions for simple, one-line functions.
- Being aware of the differences in behavior compared to regular functions, such as the lack of an "arguments" object and the different behavior of "this".
- Using curly braces and the "return" keyword for longer, more complex functions.
- Being consistent in your use of Arrow functions throughout your codebase.

**Anonymous function:**

1. What are Anonymous functions in JavaScript?

   Anonymous functions are functions that do not have a name. They can be declared on the fly and passed as arguments to other functions, or used as callbacks in events.

2. How do Anonymous functions differ from Named functions in JavaScript?
   Named functions have a name and can be used anywhere in the code. Anonymous functions do not have a name and can only be used in the context in which they are defined.

3. What is the syntax for declaring an Anonymous function in JavaScript?

   An Anonymous function can be declared using the following syntax:

```
function() {
  // code to be executed
}
```

4. How do you call an Anonymous function in JavaScript?

   An Anonymous function can be called by using the parentheses operator after the function definition:

```
(function() {
  // code to be executed
})();
```

5. Can Anonymous functions be used as callback functions in JavaScript?

   Yes, Anonymous functions can be used as callback functions in JavaScript. Here's an example:

```
document.addEventListener('click', function() {
  console.log('Button clicked!');
});
```

6. How do you pass parameters to an Anonymous function?

   You can pass parameters to an Anonymous function by including them in the parentheses after the function definition:

```
(function(param1, param2) {
  // code to be executed
})(value1, value2);
```

7. What is the difference between an Immediately Invoked Function Expression (IIFE) and a regular Anonymous function? An IIFE is an Anonymous function that is immediately called after it is defined. Here's an example:

```
(function() {
  console.log('This function is immediately called!');
})();
```

   A regular Anonymous function is just a function that is not named.

8. Can Anonymous functions have return statements?

   Yes, Anonymous functions can have return statements. Here's an example:

```
const myFunction = function() {
  return 'This function returns a value.';
};
```

9. How can you use an Anonymous function to create a closure in JavaScript? An Anonymous function can be used to create a closure in JavaScript by wrapping a function inside another function. Here's an example:

```
function outerFunction() {
  const name = 'John';
  return function() {
    console.log('Hello ' + name + '!');
  };
}

const innerFunction = outerFunction();
innerFunction();
```

In this example, the Anonymous function is created inside the outerFunction() and has access to the variable "name" even after the outer function has returned.

10. What are some best practices for using Anonymous functions in JavaScript?
Some best practices for using Anonymous functions in JavaScript include:

- Using Anonymous functions for small, one-off tasks.
- Being aware of the scope chain and potential issues with variable hoisting.
- Using descriptive variable names for Anonymous functions to improve code readability.
- Being consistent with your use of Anonymous functions throughout your codebase.

**Higher order function:**

1. What are Higher Order Functions in JavaScript?

   Higher Order Functions are functions that operate on other functions by taking them as arguments, returning them as values, or both.

2. What is the difference between a Higher Order Function and a regular function in JavaScript?

   A Higher Order Function is a function that takes another function as an argument or returns another function as its result. A regular function does not take another function as an argument or return another function as its result.

3. What are some examples of Higher Order Functions in JavaScript? Some examples of Higher Order Functions in JavaScript include:

- Array.prototype.map(): Transform and return new array.
- Array.prototype.filter(): Filter and return new array.
- Array.prototype.reduce(): Reduce to single value.
- Function.prototype.bind(): Bind context and arguments.
- Function.prototype.call(): Call with context and arguments.
- Function.prototype.apply(): Call with array of arguments.

4. What are Callback Functions and how are they related to Higher Order Functions?

   A Callback Function is a function that is passed as an argument to another function and is executed inside that function. Callback functions are related to Higher Order Functions because they are often used as arguments to Higher Order Functions.

5. What is Function Composition and how is it related to Higher Order Functions?

   Function Composition is the process of combining two or more functions to form a new function. Higher Order Functions are related to Function

Composition because they can be used to create new functions by combining existing functions.

6. What is the difference between a Pure Function and an Impure Function?

   A Pure Function is a function that always returns the same output for a given input and does not have any side effects. An Impure Function is a function that has side effects or does not always return the same output for a given input.

7. How can Higher Order Functions be used to create Pure Functions in JavaScript?

   Higher Order Functions can be used to create Pure Functions in JavaScript by abstracting away impure operations, such as I/O or state management, into separate functions and passing them as arguments to the Pure Function.

8. What are some advantages of using Higher Order Functions in JavaScript?
   Some advantages of using Higher Order Functions in JavaScript include:

- Code reusability and modularity
- Improved readability and maintainability
- Enhanced flexibility and composability

9. What are some potential disadvantages of using Higher Order Functions in JavaScript?

   Some potential disadvantages of using Higher Order Functions in JavaScript include:

- Increased complexity and cognitive load
- Decreased performance due to additional function calls
- Potential for unintended side effects or bugs

10. Can you give an example of using Higher Order Functions to solve a real-world problem in JavaScript?

    One example of using Higher Order Functions to solve a real-world problem in JavaScript is to create a function that returns the average rating of a list of products. This can be achieved using the Array.prototype.reduce() method as a Higher Order Function:

```
const products = [
  { name: 'Product 1', rating: 4.5 },
  { name: 'Product 2', rating: 3.2 },
  { name: 'Product 3', rating: 2.7 },
  { name: 'Product 4', rating: 4.0 }
];

const averageRating = products.reduce((sum, product) => sum + product.rating, 0)
/ products.length;

console.log(averageRating);
// Output: 3.6
```

**For Each:**

1. What is the purpose of the forEach() method in JavaScript?
    - The purpose of the forEach() method is to iterate over each element of an array and execute a callback function for each element.
2. What is the syntax of the forEach() method?
    - The syntax of the forEach() method is:

```
array.forEach(function(currentValue, index, array) {
  // Code to execute for each element
});
```

3. What is the difference between a forEach() loop and a regular for loop?
    - The main difference between a forEach() loop and a regular for loop is that the forEach() loop is specifically designed to iterate over each

element of an array, whereas a for loop can iterate over any sequence of values.

4. How does the forEach() method work with an array of objects?
    - The forEach() method works with an array of objects by allowing you to access each object's properties with dot notation or bracket notation within the callback function.
5. How can you break out of a forEach() loop before it has finished iterating over all the elements?
    - You cannot break out of a forEach() loop before it has finished iterating over all the elements. To accomplish this, you would need to use a regular for loop or some other loop control statement.
6. How can you use the forEach() method to modify each element of an array in place?
    - You can use the forEach() method to modify each element of an array in place by assigning a new value to each element within the callback function.
7. How can you use the forEach() method to iterate over a NodeList in the DOM?
    - You can use the forEach() method to iterate over a NodeList in the DOM by first converting it to an array using the Array.from() method, and then calling forEach() on the resulting array.
8. Can the forEach() method be used with asynchronous code? If so, how?
    - Yes, the forEach() method can be used with asynchronous code by using promises or async/await syntax within the callback function.
9. How can you pass additional arguments to the callback function of the forEach() method?
    - You can pass additional arguments to the callback function of the forEach() method by specifying them after the index parameter, separated by commas.

10. What is the return value of the forEach() method?
    - The return value of the forEach() method is undefined.

**Set TimeOut:**

1. What is the purpose of the setTimeout() method in JavaScript?
    - The purpose of the setTimeout() method is to schedule a function to execute after a specified amount of time has elapsed.
2. What is the syntax of the setTimeout() method?
    - The syntax of the setTimeout() method is:

```
setTimeout(function() {
  // Code to execute after a delay
}, delay);
```

3. How does the setTimeout() method work with a callback function?
    - The setTimeout() method works with a callback function by scheduling the function to be executed after the specified delay has elapsed.
4. How do you cancel a setTimeout() function before it executes?
    - You can cancel a setTimeout() function before it executes by calling the clearTimeout() method and passing in the timeout ID returned by the original setTimeout() call.
5. How can you pass arguments to the callback function of a setTimeout() method?
    - You can pass arguments to the callback function of a setTimeout() method by specifying them after the delay parameter, separated by commas.
6. How can you use the setTimeout() method to create a repeating timer?

- You can use the setTimeout() method to create a repeating timer by recursively calling setTimeout() from within the callback function and passing in the desired delay time.

7. What is the difference between setTimeout() and setInterval()?
    - The main difference between setTimeout() and setInterval() is that setTimeout() executes a function once after a delay, while setInterval() executes a function repeatedly at a specified interval until stopped.
8. What is the minimum and maximum delay time for setTimeout()?
    - The minimum delay time for setTimeout() is typically 4 milliseconds, although this can vary depending on the browser or environment. There is no maximum delay time.
9. How can you measure the performance of a function using setTimeout()?
    - You can measure the performance of a function using setTimeout() by recording the start time before calling the function and the end time inside the callback function, and then subtracting the start time from the end time to calculate the elapsed time.
10. Can the setTimeout() method be used with asynchronous code? If so, how?
    - Yes, the setTimeout() method can be used with asynchronous code by using promises or async/await syntax within the callback function.

**Set Interval:**

1. What is the purpose of the setInterval() method in JavaScript?
    - The purpose of the setInterval() method is to repeatedly execute a function at a specified time interval.
2. What is the syntax of the setInterval() method?
    - The syntax of the setInterval() method is:

```
setInterval(function() {
  // Code to execute at each interval
}, interval);
```

3. How does the setInterval() method work?
    - The setInterval() method schedules a function to be executed at a specified time interval. It continues to execute the function repeatedly until stopped.
4. How do you stop a setInterval() function from executing?
    - You can stop a setInterval() function from executing by calling the clearInterval() method and passing in the interval ID returned by the original setInterval() call.
5. How can you pass arguments to the callback function of a setInterval() method?
    - You can pass arguments to the callback function of a setInterval() method by specifying them after the function parameter, separated by commas.
6. How can you use the clearInterval() method to stop a repeating timer?
    - You can use the clearInterval() method to stop a repeating timer by passing in the interval ID returned by the original setInterval() call.
7. What is the minimum and maximum delay time for setInterval()?

- The minimum delay time for setInterval() is typically 10 milliseconds, although this can vary depending on the browser or environment. There is no maximum delay time.

8. What is the difference between setTimeout() and setInterval()?
    - The main difference between setTimeout() and setInterval() is that setTimeout() executes a function once after a delay, while setInterval() executes a function repeatedly at a specified interval until stopped.
9. How can you measure the performance of a function using setInterval()?
    - You can measure the performance of a function using setInterval() by recording the start time before calling the function and the end time inside the callback function, and then subtracting the start time from the end time to calculate the elapsed time.
10. Can the setInterval() method be used with asynchronous code? If so, how?
    - Yes, the setInterval() method can be used with asynchronous code by using promises or async/await syntax within the callback function.

**Asynchronous Programming:**

1. What is asynchronous programming in JavaScript?
    - Asynchronous programming in JavaScript allows for code to execute out of order and to continue running while waiting for long-running operations to complete.
2. What is the difference between synchronous and asynchronous programming?
    - Synchronous programming executes code in order, one line at a time, while asynchronous programming allows code to continue executing while waiting for long-running operations to complete.

3. What are some common examples of asynchronous operations in JavaScript?
    - Some common examples of asynchronous operations in JavaScript include making HTTP requests, reading or writing to a file, and processing large amounts of data.
4. How do you handle errors with asynchronous code in JavaScript?
    - Errors with asynchronous code in JavaScript can be handled using try/catch statements or by attaching an error callback function to the asynchronous operation.
5. What is a callback function in JavaScript?
    - A callback function in JavaScript is a function that is passed as an argument to another function and is executed after the completion of that function.
6. What is a Promise in JavaScript?
    - A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and allows for more sophisticated error handling and chaining of multiple operations.
7. What is the async/await syntax in JavaScript?
    - The async/await syntax in JavaScript is a way to write asynchronous code that looks and behaves more like synchronous code, by allowing for the use of the **await** keyword to pause execution until a Promise resolves or rejects.
8. How do you create a Promise in JavaScript?
    - A Promise in JavaScript can be created using the Promise constructor, which takes a function with **resolve** and **reject** arguments that can be called to either fulfill or reject the Promise. Example syntax:

```
const myPromise = new Promise((resolve, reject) => {
  // some asynchronous operation
  if (success) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

9. How can you chain multiple asynchronous operations together in JavaScript?

    - Multiple asynchronous operations in JavaScript can be chained together using Promise methods like **then** and **catch**, or by using the async/await syntax to wait for each operation to complete before moving on to the next. Example syntax:

```
myPromise
  .then(result => {
    // handle success
    return anotherPromise;
  })
  .then(anotherResult => {
    // handle success
  })
  .catch(error => {
    // handle error
  });
```

10. How do you handle concurrency with asynchronous code in JavaScript?

    - Concurrency with asynchronous code in JavaScript can be handled using techniques like parallelism, throttling, and queuing to balance performance with resource usage and prevent race conditions. Example syntax:

```
// Parallelism example
Promise.all([promise1, promise2, promise3])
  .then(results => {
    // handle success
  })
  .catch(error => {
    // handle error
  });

// Throttling example
let i = 0;
const promises = [];
while (i < 10) {
  promises.push(new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(result);
    }, 1000);
  }));
  i++;
}
Promise.all(promises)
  .then(results => {
    // handle success
  })
  .catch(error => {
    // handle error
  });
```

**Fetch API:**

1. What is the fetch API in JavaScript?

- The fetch API is a built-in JavaScript interface for making asynchronous network requests to servers using HTTP or HTTPS.

2. What is the syntax for making a simple GET request using the fetch API?

- The syntax for making a simple GET request using the fetch API is:

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, **url** is the URL of the resource that you want to fetch.

3. What does the **fetch()** function return?

- The **fetch()** function returns a promise that resolves to the response from the server. The response object contains information such as the status of the response, headers, and the response body.

4. What are the HTTP methods that can be used with the fetch API?

- The fetch API can be used with any HTTP method, including GET, POST, PUT, DELETE, and others.

5. How can you add custom headers to a fetch request?

- You can add custom headers to a fetch request by passing an options object as the second parameter to the fetch function. For example:

```
fetch(url, {
  headers: {
    'Authorization': 'Bearer <token>',
    'Content-Type': 'application/json'
  }
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

6. How do you handle errors when making a fetch request?

- You can handle errors when making a fetch request by chaining a **catch()** method to the end of the promise chain. For example:

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, if the network request fails or the server returns an error response, the **catch()** method will be called with the error object.

7. What is the difference between the **fetch()** API and the traditional **XMLHttpRequest** API?

- The **fetch()** API is a newer, more modern interface for making network requests that is designed to be easier to use and more flexible than the **XMLHttpRequest** API. The **fetch()** API uses promises and is built around a simple request-response model, whereas the **XMLHttpRequest** API is a more complex and lower-level interface that can be more difficult to use.