1. What is ES6? Have you ever used anything from ES6?

Answer: JavaScript ES6 brings new syntax and new awesome features to make your code more modern and more readable. It allows you to write less code and do more. ES6 introduces us to many great features like arrow functions, template strings, class destruction, Modules… and more.

https://www.freecodecamp.org/news/write-less-do-more-with-javascript-es6-5fd4a8e50ee2/

2. Explain the difference between **var, let** and **const**.

Answer: `var`, `let`, and `const` are all used to declare variables in JavaScript, but they have different scoping and mutability characteristics:

- **`var`**:
  - Variables declared with `var` are function-scoped, meaning they are accessible throughout the entire function in which they are defined, regardless of block scope.
  - They are also hoisted to the top of their containing function or global scope, which means you can access the variable before it's declared.
  - `var` variables can be redeclared within the same scope without generating errors.
  - They can also be updated and reassigned.

- **`let`**:
  - Variables declared with `let` are block-scoped, meaning they are only accessible within the block of code in which they are defined.
  - Like `var`, `let` variables are hoisted, but they are not initialized until the code execution reaches their declaration.
  - Unlike `var`, `let` variables cannot be redeclared in the same scope, which helps catch errors.
  - They can be updated and reassigned.

- **`const`**:
  - Variables declared with `const` are also block-scoped.
  - The key difference is that `const` variables cannot be reassigned after their initial assignment. They are read-only.

- However, for objects and arrays assigned to `const`, their properties or elements can still be modified. The immutability only applies to the reference itself, not the contents.

- Like `let`, `const` variables are also not hoisted and are only available in the block where they are defined.

3. What is the Arrow function and How to create it?

Answer: An arrow function is a concise way to write a function expression in JavaScript. It was introduced in ES6 (ECMAScript 2015) and provides a more compact syntax compared to traditional function expressions. Arrow functions are often used for writing shorter and more readable code, especially when dealing with functions that have simple logic.

Here's the syntax for creating an arrow function:

```
const functionName = (parameters) => {
  // function body
  // return statement (if applicable)
};
```

4. Give an example of an Arrow function in ES6? List down it's advantages.

Answer: Example of an Arrow Function:

JavaScript:

Copy code

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

// Using arrow function to calculate the square of each number
const squaredNumbers = numbers.map((num) => num * num);

console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

Advantages of Arrow Functions:

Concise Syntax: Arrow functions provide a more concise syntax for writing functions, especially when the logic is simple. This can make your code cleaner and easier to read.

Implicit Return: Arrow functions with a single expression can have an implicit return, which means you don't need to explicitly use the return keyword. This further reduces code verbosity.

Lexical this Binding: Arrow functions do not have their own this context. Instead, they inherit the this value from the surrounding code. This can help avoid the common confusion and complexities associated with the this keyword in regular function expressions.

No Function Name: Arrow functions are always anonymous, meaning they don't have a named identifier. This can be useful when creating short, simple utility functions that don't require a specific name.

No arguments Binding: Unlike traditional function expressions, arrow functions do not have their own arguments object. This can lead to more predictable behavior when dealing with function arguments.

Compatibility with Functional Programming: Arrow functions align well with functional programming principles and are commonly used with array methods like map, filter, and reduce.

Readability: The concise syntax and reduced need for explicit code (such as function and return keywords) often lead to improved code readability.

Closure Behavior: Arrow functions capture variables from their containing scope, which can be useful in certain scenarios where you want to maintain access to external variables.

Here's how the example above demonstrates some of these advantages:

The arrow function used in the map method is concise and requires minimal syntax.

The implicit return in the arrow function allows for a cleaner expression.

The arrow function inherits the correct this context from the surrounding code, making it easier to work with.

However, it's important to note that arrow functions are not a replacement for traditional functions in all cases. They are most effective for short, simple functions and may not be suitable for functions with more complex logic or when you need access to the arguments object.

5. Discuss spread operator in ES6 with an example?

Answer: The spread operator (`...`) is a powerful feature introduced in ES6 (ECMAScript 2015) that allows you to expand elements of an iterable (like an array, string, or object) into various contexts, such as function arguments, array literals, or object literals. The spread operator is a versatile tool that makes working with data in JavaScript more flexible and concise.

Here's an example of how the spread operator works with arrays:

JavaScript:

const originalArray = [1, 2, 3];

const newArray = [...originalArray, 4, 5];

console.log(newArray); // Output: [1, 2, 3, 4, 5]

```

In this example, the spread operator `...originalArray` expands the elements of the `originalArray` into the new array `newArray`. The result is that the new array contains all the elements from the original array along with the additional elements `[4, 5]`.

The spread operator can be used in various contexts:

1. **Copying Arrays**:

   You can use the spread operator to create a shallow copy of an array:

   ```javascript

   const originalArray = [1, 2, 3];

```
const copiedArray = [...originalArray];

console.log(copiedArray); // Output: [1, 2, 3]
```

2. **Concatenating Arrays**:

   You can concatenate arrays using the spread operator:

   ```javascript
   const array1 = [1, 2, 3];
   const array2 = [4, 5];
   const concatenatedArray = [...array1, ...array2];

   console.log(concatenatedArray); // Output: [1, 2, 3, 4, 5]
   ```

3. **Passing Array Elements as Function Arguments**:

   You can use the spread operator to pass array elements as individual arguments to a function:

   ```javascript
   const numbers = [1, 2, 3];
   const sum = (a, b, c) => a + b + c;

   const result = sum(...numbers); // Equivalent to sum(1, 2, 3)

   console.log(result); // Output: 6
   ```

4. **Creating New Arrays with Modified Elements**:

   You can modify specific elements of an array while creating a new array:

   ```javascript
   ```

```
const originalArray = [1, 2, 3];
const modifiedArray = [...originalArray.slice(0, 2), 4, 5];

console.log(modifiedArray); // Output: [1, 2, 4, 5]
```

5. **Using with Object Literals**:

   You can use the spread operator to merge properties of objects into a new object:

   ```javascript
   const person = { name: "Alice", age: 30 };
   const additionalInfo = { location: "New York", occupation: "Engineer" };

   const mergedPerson = { ...person, ...additionalInfo };

   console.log(mergedPerson); // Output: { name: 'Alice', age: 30, location: 'New York', occupation: 'Engineer' }
   ```

The spread operator is a powerful tool that simplifies various operations in JavaScript, making code more concise and expressive. However, keep in mind that the spread operator performs a shallow copy, meaning nested objects or arrays will still be referenced, not duplicated.

6. What do you understand about default parameters?

Answer: Default parameters are a feature introduced in ES6 (ECMAScript 2015) that allow you to assign default values to function parameters in case the caller does not provide a value for those parameters or provides `undefined`.

Before default parameters, developers often had to use conditional statements within functions to handle missing or undefined values for parameters. Default parameters simplify this process by allowing you to set default values directly in the function declaration.

Here's the syntax for using default parameters in a function:

```javascript
function functionName(parameter = defaultValue) {
  // function body
}
```

Here's an example to illustrate the concept:

```javascript
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}

greet(); // Output: Hello, Guest!
greet("Alice"); // Output: Hello, Alice!
```

In this example, the `name` parameter of the `greet` function has a default value of `"Guest"`. If the caller does not provide an argument for `name`, the default value will be used instead.

Key points to note about default parameters:

1. **Default Values**: Default parameters allow you to specify a default value that will be used if the parameter is not explicitly provided by the caller or is `undefined`.

2. **Position Matters**: Default parameters are assigned in the order they appear in the parameter list. You cannot skip a parameter and provide a default value for a parameter later in the list.

3. **Evaluation**: The default value expression is evaluated at the time the function is called, not at the time of function declaration. This can lead to dynamic default values based on the function's context.

4. **Default Value Precedence**: If the caller provides an argument (even if it's `null`), the provided value takes precedence over the default value.

5. **Default Values and Undefined**: The default value is used only if the parameter is missing or explicitly passed as `undefined`. If the parameter is passed as `null`, `false`, `0`, or an empty string, the default value will not be applied.

6. **Usage in Functions**: Default parameters are not limited to just primitive values. They can also be used with expressions, objects, arrays, and even function calls.

Here's an example with an object default parameter:

```javascript
function createUser(info = { name: "Guest", age: 0 }) {
  console.log(info);
}

createUser(); // Output: { name: 'Guest', age: 0 }
createUser({ name: "Alice", age: 30 }); // Output: { name: 'Alice', age: 30 }
```

Default parameters provide a more elegant way to handle function parameters with fallback values, leading to cleaner and more maintainable code.

7. What are template literals in ES6?

Answer: Template literals, also known as template strings, are a feature introduced in ES6 (ECMAScript 2015) that allow you to create strings with embedded expressions in a more readable and convenient way compared to traditional string concatenation.

Template literals are enclosed in backticks (`` ` ``) instead of single or double quotes. Inside a template literal, you can embed placeholders for expressions using the `${expression}` syntax. When the template literal is evaluated, the expressions inside the placeholders are evaluated and their results are inserted into the string.

Here's an example of using template literals:

JavaScript:

```
const name = "Alice";

const age = 30;

const message = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(message);

// Output: Hello, my name is Alice and I am 30 years old.
```

In this example, the variables `name` and `age` are embedded within the template literal using `${}` placeholders. When the template literal is evaluated, the values of these variables are inserted into the string, resulting in a more dynamic and readable output.

Key features of template literals:

1. **Multi-line Strings**: Template literals preserve line breaks, allowing you to create multi-line strings without needing explicit escape characters or concatenation.

2. **Expression Embedding**: You can embed any valid JavaScript expression within `${}` placeholders. This can include variables, calculations, function calls, and more.

3. **String Formatting**: Template literals provide a clean way to format strings with dynamic content, making code more concise and readable.

4. **Tagged Templates**: Template literals can also be used with a feature called "tagged templates," where you can apply a function to the template literal. This can be used for custom string manipulation or localization.

Here's an example of a tagged template:

```javascript
function customTag(strings, ...values) {
  return strings[0] + values[0] + strings[1] + values[1];
}

const a = 5;
const b = 10;
const result = customTag`Sum of ${a} and ${b} is ${a + b}.`;

console.log(result);
// Output: Sum of 5 and 10 is 15.
```

Template literals are a powerful and flexible feature that improves the readability and maintainability of string manipulation in JavaScript. They are especially useful when creating dynamic strings that incorporate variables and expressions.

8. Tell us the difference between arrow and regular function.

Answer: Arrow functions and regular (traditional) functions in JavaScript have differences in terms of syntax, behavior, and how they handle the `this` keyword. Let's explore these differences:

1. Syntax:

Regular Function:

```javascript
function regularFunction(param) {
  // function body
}
```

Arrow Function:

```javascript
const arrowFunction = (param) => {
  // function body
};
```

Arrow functions have a more concise syntax, especially when the function body is a single expression. If the function body is a single statement, you can even omit the curly braces and the `return` keyword.

2. `this` Binding:

Regular Function:

Regular functions have their own `this` context, which can change depending on how the function is called. In methods or callback functions, `this` often refers to the object that the method belongs to.

Arrow Function:

Arrow functions do not have their own `this` context. Instead, they inherit the `this` value from the enclosing scope (lexical `this` binding). This can be advantageous in avoiding confusion and unexpected behavior related to `this`.

3. Arguments Object:

Regular Function:

Regular functions have access to the `arguments` object, which holds all the arguments passed to the function, regardless of the number of declared parameters.

Arrow Function:

Arrow functions do not have their own `arguments` object. They inherit the `arguments` object from the containing non-arrow function if used within one.

4. Constructor:

Regular Function:

Regular functions can be used as constructors to create new instances of objects using the `new` keyword.

Arrow Function:

Arrow functions cannot be used as constructors. They do not have their own `this`, which is required for constructor functions.

5. No Binding of `this`, `super`, `new.target`, and `arguments`:

Arrow functions do not have their own bindings for `this`, `super`, `new.target`, and `arguments`. These values are inherited from the containing function or scope.

6. Use Cases:

Regular functions are suitable for most scenarios and offer more flexibility in terms of `this` binding and access to the `arguments` object.

Arrow functions are best used when concise syntax is desired, especially for short, simple functions. They are also useful when you want to capture the value of `this` from the surrounding context without worrying about it changing.

In summary, regular functions provide more flexibility in terms of `this` binding and handling of `arguments`, while arrow functions offer a shorter syntax and inherent lexical `this` binding. The choice between the two depends on the specific use case and the behavior you need for your functions.

9. Tell us the difference between seal and freeze.

Answer: In JavaScript, both `Object.seal()` and `Object.freeze()` are methods used to restrict the modification of objects, but they provide different levels of immutability and protection. Let's explore the differences between the two:

1. Object.seal():

`Object.seal()` is a method that seals an object, preventing the addition of new properties and making existing properties non-configurable. However, it does allow the modification of existing property values.

Key characteristics of a sealed object:

- Properties can be modified if they were writable before sealing.

- Properties cannot be deleted.

- New properties cannot be added.

- Property attributes (like configurability and writability) cannot be changed.

Example using `Object.seal()`:

JavaScript:

```
const sealedObject = {
```

```
  prop1: 10,
  prop2: "hello"
};

Object.seal(sealedObject);

// Modifying existing property values is allowed
sealedObject.prop1 = 20;

// Adding new properties is not allowed
sealedObject.prop3 = "new property"; // Will have no effect

// Deleting properties is not allowed
delete sealedObject.prop2; // Will have no effect

console.log(sealedObject); // Output: { prop1: 20, prop2: 'hello' }
```

2. Object.freeze():

`Object.freeze()` goes a step further than `Object.seal()`. It not only seals the object but also makes the properties read-only, preventing any modification of property values.

Key characteristics of a frozen object:

- All the characteristics of a sealed object.
- Properties are also made read-only.

Example using `Object.freeze()`:

```javascript
const frozenObject = {
```

```
  prop1: 10,
  prop2: "hello"
};

Object.freeze(frozenObject);

// Modifying existing property values is not allowed
frozenObject.prop1 = 20; // Will have no effect

// Adding new properties is not allowed
frozenObject.prop3 = "new property"; // Will have no effect

// Deleting properties is not allowed
delete frozenObject.prop2; // Will have no effect

console.log(frozenObject); // Output: { prop1: 10, prop2: 'hello' }
```

Key Differences:

1. **Property Mutability**:
   - `Object.seal()` allows modification of existing property values.
   - `Object.freeze()` makes properties both non-configurable and read-only, preventing any modification.

2. **Addition and Deletion of Properties**:
   - Neither `Object.seal()` nor `Object.freeze()` allows adding new properties or deleting existing ones.

3. **Read-only Properties**:

- Only `Object.freeze()` makes properties read-only in addition to other restrictions.

4. **Use Cases**:

- Use `Object.seal()` when you want to prevent the addition of new properties and make existing properties non-configurable, but still allow property value changes.

- Use `Object.freeze()` when you want to completely prevent any modification to an object, including making properties read-only.

Both `Object.seal()` and `Object.freeze()` are useful tools for maintaining data integrity and preventing accidental modifications to objects. The choice between them depends on the level of protection you need for your objects.

10. Tell us the difference between for…of and for…in.

Answer: Both `for...of` and `for...in` are used to iterate over elements in JavaScript, but they are used in different contexts and have different behaviors. Let's explore the differences between these two loop constructs:

1. `for...of` Loop:

The `for...of` loop is used to iterate over the values of iterable objects, such as arrays, strings, maps, sets, and more. It provides a simple and concise way to iterate over the elements without the need for index tracking.

Example using `for...of` loop:

JavaScript:

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

for (const num of numbers) {
  console.log(num);
}
// Output:
// 1
```

```
// 2
// 3
// 4
// 5
```

Key Characteristics of `for...of` Loop:

- Iterates over values.
- Works with iterable objects.
- Provides direct access to each value without index.

2. `for...in` Loop:

The `for...in` loop is used to iterate over the enumerable properties of an object, including its prototype chain. It's commonly used for objects and is particularly suited for iterating over keys or properties.

Example using `for...in` loop:

```javascript
const person = {
  name: "Alice",
  age: 30,
  occupation: "Engineer"
};

for (const key in person) {
  console.log(key, person[key]);
}
// Output:
// name Alice
```

// age 30

// occupation Engineer

```

**Key Characteristics of `for...in` Loop:**

- Iterates over property names (keys).

- Works with objects and iterates through enumerable properties.

- Can also iterate through properties inherited from the prototype chain (use `hasOwnProperty()` to filter them out).

**Differences:**

1. **Iterating Over**:

   - `for...of` iterates over values of iterable objects.

   - `for...in` iterates over property names (keys) of objects.

2. **Order**:

   - `for...of` maintains the order of iteration based on the order of elements in the iterable.

   - `for...in` does not guarantee a specific order and may iterate over properties in an arbitrary order.

3. **Inherited Properties**:

   - `for...of` does not iterate through inherited properties.

   - `for...in` can iterate through properties inherited from the prototype chain. To avoid this, it's common to use `hasOwnProperty()` to filter out inherited properties.

4. **Usage Context**:

   - Use `for...of` when you want to iterate over values of iterable objects.

   - Use `for...in` when you want to iterate over keys or properties of objects, and be cautious about inherited properties.
```

5. **String Iteration**:

   - `for...of` iterates over individual characters in strings.

   - `for...in` iterates over the indices (keys) of characters in strings.

In summary, use the `for...of` loop when you want to iterate over values in iterable objects like arrays or strings, and use the `for...in` loop when you want to iterate over properties of objects (keys), being aware of potential inherited properties from the prototype chain.