1. What’s the difference between map, forEach, filter?

Answer: `map`, `forEach`, and `filter` are three commonly used array methods in JavaScript, each serving a different purpose when it comes to working with arrays. Let's discuss the differences between these methods:

1. `map()` Method:

The `map()` method creates a new array by applying a provided function to every element in the calling array. It returns a new array with the same length as the original array, where each element is the result of the function applied to the corresponding element of the original array.

Example using `map()`:

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

const squaredNumbers = numbers.map(num => num * num);

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

Key Characteristics of `map()` Method:

- Creates a new array with transformed elements.
- Returns an array with the same length as the original array.

2. `forEach()` Method:

The `forEach()` method iterates over each element in the array and applies a provided function to each element. Unlike `map()`, `forEach()` does not create a new array; it's mainly used for executing side effects, like updating existing elements or performing actions for each element.

Example using `forEach()`:

```javascript
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(num => {
  console.log(num * 2);
});
// Output:
// 2
// 4
// 6
// 8
// 10
```

Key Characteristics of `forEach()` Method:

- Iterates over array elements.
- Does not create a new array; mainly used for side effects.
- The return value of the callback function within `forEach()` is ignored.

3. `filter()` Method:

The `filter()` method creates a new array containing all elements that pass a certain condition specified by a provided function. It returns a new array with only the elements that satisfy the given condition.

Example using `filter()`:

```javascript
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers); // Output: [2, 4]
```

```

**Key Characteristics of `filter()` Method:**

- Creates a new array containing elements that meet a specific condition.

- Returns a new array that may have fewer elements than the original array.

**Differences:**

1. **Purpose**:

  - `map()` transforms each element in an array and returns a new array.

  - `forEach()` iterates over elements for side effects (e.g., logging, updating values).

  - `filter()` creates a new array with elements that meet a condition.

2. **Return Value**:

  - `map()` returns a new array with transformed elements.

  - `forEach()` does not return anything; it's used for side effects.

  - `filter()` returns a new array with filtered elements.

3. **Modifying Original Array**:

  - `map()` and `filter()` do not modify the original array; they create new arrays.

  - `forEach()` does not create a new array, but it can be used to modify existing elements.

4. **Callback Function Usage**:

  - The callback function provided to `map()` and `filter()` should return a value.

  - The callback function provided to `forEach()` is mainly used for side effects.

In summary, `map()`, `forEach()`, and `filter()` are powerful array methods that serve different purposes. Use `map()` when you want to transform elements, `forEach()` for side effects, and `filter()` to create a new array with elements that meet a specific condition.
```

2. What’s the difference between filter and find?

Answer: `filter()` and `find()` are both array methods in JavaScript that deal with searching for specific elements based on certain conditions. However, they have different purposes and behaviors. Let's explore the differences between these two methods:

1. `filter()` Method:

The `filter()` method creates a new array containing all elements that meet a specific condition specified by a provided function. It iterates through all elements of the array and returns a new array containing all elements that satisfy the given condition.

Example using `filter()`:

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

const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers); // Output: [2, 4]
```

Key Characteristics of `filter()` Method:

- Creates a new array containing elements that meet a specific condition.
- Returns a new array that may have fewer elements than the original array.

2. `find()` Method:

The `find()` method returns the first element in the array that meets a specific condition specified by a provided function. It stops as soon as it finds the first element that satisfies the condition and returns that element. If no element satisfies the condition, `undefined` is returned.

Example using `find()`:

```javascript
```

```
const numbers = [1, 2, 3, 4, 5];
const firstEvenNumber = numbers.find(num => num % 2 === 0);

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

Key Characteristics of `find()` Method:

- Returns the first element that meets a specific condition.
- Stops iterating once a satisfying element is found.
- Returns `undefined` if no satisfying element is found.

Differences:

1. **Returned Value**:
   - `filter()` returns a new array containing all elements that meet a condition.
   - `find()` returns the first element that meets a condition or `undefined` if no element meets the condition.

2. **Use Case**:
   - Use `filter()` when you want to create a new array with multiple elements that meet a specific condition.
   - Use `find()` when you want to find the first element that meets a specific condition.

3. **Array Length**:
   - `filter()` may return an array with multiple elements.
   - `find()` returns a single element or `undefined`.

4. **Stopping Behavior**:
   - `filter()` iterates through all elements regardless of whether a condition is met or not.
   - `find()` stops iterating as soon as it finds the first element that meets the condition.

5. **Performance Consideration**:

- If you only need to find one element that meets a condition, using `find()` might be more efficient as it stops iterating once that element is found.

- If you need to collect multiple elements that meet a condition, `filter()` is more suitable.

In summary, while both `filter()` and `find()` are used for searching elements in arrays, they have different behaviors and are used in different contexts. Use `filter()` to create a new array with multiple elements that meet a condition, and use `find()` to find the first element that meets a condition.

3. What's the difference between for…of and for…in?

Answer: Both `for...of` and `for...in` are loop constructs in JavaScript used for iteration, but they have different purposes and behaviors. Let's explore the differences between these two loop types:

1. `for...of` Loop:

The `for...of` loop is used to iterate over values of iterable objects, such as arrays, strings, maps, sets, and more. It provides a simple and concise way to iterate over the elements of an iterable 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 of iterable objects.

- Works with arrays, strings, maps, sets, and other iterables.

- 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) of objects.

- 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 like arrays or strings.

  - 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, `for...of` and `for...in` are used for different purposes: `for...of` is used to iterate over iterable values, while `for...in` is used to iterate over object properties. Consider the nature of the data you're working with when choosing between these two loop types.

4. How do you empty an array?

Answer: To empty an array in JavaScript, you can use a few different methods depending on your use case. Here are three common approaches:

1. Setting Length to 0:

One straightforward way to empty an array is by setting its length to 0. This effectively removes all elements from the array.

```javascript
const array = [1, 2, 3, 4, 5];
array.length = 0;

console.log(array); // Output: []
```

2. Using the splice() Method:

The `splice()` method can be used to remove elements from an array starting from a specified index. If you specify an index of 0 and a count of the array's length, it will remove all elements from the array.

```javascript
```

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

array.splice(0, array.length);


console.log(array); // Output: []
```

3. Reassigning with an Empty Array:

Another approach is to simply reassign the variable holding the array to a new, empty array.

```javascript
let array = [1, 2, 3, 4, 5];

array = [];


console.log(array); // Output: []
```

It's important to note that these methods will directly modify the original array and remove all of its elements. If you have references to the original array elsewhere in your code, they will also reflect the changes.

Choose the method that best fits your use case and coding style.

5. Difference between class and object.

Answer: In object-oriented programming, both classes and objects are fundamental concepts, but they serve different purposes and have distinct roles. Let's explore the differences between classes and objects:

1. Class:

A class is a blueprint or template for creating objects. It defines the structure and behavior that objects of the class will have. In other words, a class is like a prototype that defines the properties (attributes) and methods (functions) that objects belonging to that class will have.

Key characteristics of a class:

- Defines the attributes (properties) and methods (functions) that objects will have.

- Acts as a blueprint for creating multiple objects with the same structure.

Example of a class definition in JavaScript:

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

  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
```

2. Object:

An object is an instance of a class. It's a concrete entity that holds specific values for the attributes defined in the class and can call the methods defined in the class. Objects are created based on the structure defined by the class.

Key characteristics of an object:

- Represents a specific instance with actual values for attributes.

- Can call methods and access properties defined in its class.

Example of creating objects based on the class definition:

```javascript
const person1 = new Person("Alice", 30);
const person2 = new Person("Bob", 25);

person1.greet(); // Output: Hello, my name is Alice and I am 30 years old.
person2.greet(); // Output: Hello, my name is Bob and I am 25 years old.
```

Differences:

1. **Abstract vs. Concrete:**
   - A class is an abstract representation that defines the structure and behavior of objects.
   - An object is a concrete instance created based on a class, with actual values for attributes.

2. **Blueprint vs. Instance:**
   - A class acts as a blueprint or template for creating objects.
   - An object is a specific instance created from a class blueprint.

3. **Definition vs. Usage:**
   - A class defines the properties and methods that objects will have.
   - An object represents a specific entity and can call methods and access properties.

4. **One-to-Many Relationship:**
   - One class definition can be used to create multiple objects with similar structures and behaviors.

5. **Instantiation:**

- To use a class and create objects, you need to instantiate it using the `new` keyword.

In summary, a class is a blueprint that defines the structure and behavior of objects, while an object is a specific instance created based on that blueprint. Classes allow you to create consistent and organized code by defining the common properties and methods that objects will share.

6. What is a Prototype chain? Or how does inheritance work in JavaScript?

Answer: In JavaScript, inheritance is achieved through a mechanism known as the prototype chain. The prototype chain is a fundamental concept that allows objects to inherit properties and methods from other objects, forming a chain of prototypes. This mechanism provides a way to share and reuse code among objects.

Here's how the prototype chain and inheritance work in JavaScript:

1. Prototype Object:

Every object in JavaScript has a property called `__proto__` (dunder proto) that points to another object, known as its prototype. This prototype object itself can have its own prototype, forming a chain of objects connected through their `__proto__` properties.

2. Object Creation and Inheritance:

When you access a property or method on an object, JavaScript first checks if the object itself has that property. If not, it looks up the prototype chain, going up the chain until it finds the property or reaches the end of the chain.

Here's a simplified example:

```javascript
// Parent object
const parent = {
  parentProp: "I am a parent property",
```

};

// Child object with parent as prototype

const child = Object.create(parent);

child.childProp = "I am a child property";

console.log(child.childProp);   // Output: I am a child property

console.log(child.parentProp);   // Output: I am a parent property

```

In this example, the `child` object inherits the `parentProp` property from its prototype (`parent`), even though `child` doesn't have that property directly.

**3. Constructor Functions and Prototypes:**

In JavaScript, constructor functions are used to create objects with shared properties and methods. When a constructor function is invoked with the `new` keyword, it creates a new object, sets its prototype to the constructor function's `prototype` property, and allows you to initialize the object's properties.

```javascript

function Person(name) {

  this.name = name;

}

Person.prototype.greet = function() {

  console.log(`Hello, my name is ${this.name}`);

};

const person1 = new Person("Alice");

person1.greet(); // Output: Hello, my name is Alice

```

In this example, the `Person` constructor function defines a `greet` method on its prototype. When an instance of `Person` is created (`person1`), it inherits the `greet` method from the prototype chain.

4. Inheriting from Built-in Objects:

JavaScript's built-in objects also have prototypes, which allow you to extend their functionality. For example, you can add custom methods to the prototype of the `Array` object and have those methods available to all arrays.

```javascript
Array.prototype.customMethod = function() {
  console.log("This is a custom method for arrays");
};

const myArray = [1, 2, 3];
myArray.customMethod(); // Output: This is a custom method for arrays
```

5. Object.create() Method:

The `Object.create()` method allows you to create a new object with a specified prototype object. This is useful for creating objects with specific inheritance relationships.

```javascript
const prototypeObject = {
  sharedProp: "This is a shared property",
};

const newObj = Object.create(prototypeObject);
```
```

```
console.log(newObj.sharedProp); // Output: This is a shared property
```

In summary, the prototype chain is a mechanism that enables objects to inherit properties and methods from other objects. It forms the basis of inheritance in JavaScript and allows for code reuse, extensibility, and flexibility in object-oriented programming.

7. What does destructing do in es6?

Answer: Destructuring is a feature introduced in ES6 (ECMAScript 2015) that allows you to extract values from objects or arrays and assign them to variables in a more concise and readable way. It simplifies the process of extracting specific values from complex data structures.

Destructuring can be used for both objects and arrays. Let's explore how destructuring works for each of these data structures:

1. Destructuring Objects:

With object destructuring, you can extract values from an object and assign them to variables with the same names as the object's properties. You use curly braces `{}` to specify the variables you want to create, and you match the variable names to the property names in the object.

Example of object destructuring:

```javascript
const person = {
  firstName: "Alice",
  lastName: "Smith",
  age: 30,
};

const { firstName, lastName, age } = person;
```

console.log(firstName); // Output: Alice

console.log(lastName); // Output: Smith

console.log(age);   // Output: 30

```

**2. Destructuring Arrays:**

With array destructuring, you can extract values from an array and assign them to variables. You use square brackets `[]` to specify the variables you want to create, and the order of variables matches the order of values in the array.

Example of array destructuring:

```javascript

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

const [first, second, ...rest] = numbers;

console.log(first);  // Output: 1

console.log(second); // Output: 2

console.log(rest);   // Output: [3, 4, 5]

```

In this example, the `rest` variable uses the rest parameter syntax (`...`) to capture the remaining values in the array.

3. Default Values:

Destructuring also allows you to provide default values for variables in case the property or element being destructured is `undefined`.

Example with default values:

```javascript

```
const person = {
  firstName: "Alice",
};

const { firstName, lastName = "Doe" } = person;

console.log(firstName); // Output: Alice
console.log(lastName); // Output: Doe (default value)
```

4. Nested Destructuring:

You can also use nested destructuring to extract values from nested objects or arrays.

Example of nested object destructuring:

```javascript
const person = {
  name: {
    first: "Alice",
    last: "Smith",
  },
  age: 30,
};

const { name: { first, last }, age } = person;

console.log(first); // Output: Alice
console.log(last);  // Output: Smith
console.log(age);   // Output: 30
```

Destructuring is a powerful and concise way to extract values from complex data structures, making your code more readable and less verbose. It's commonly used in situations where you need to work with specific values from objects or arrays.

8. Is optional chaining is same as ternary operator?

Answer: No, optional chaining and the ternary operator serve different purposes and have different use cases in JavaScript.

Optional Chaining (`?.`):

Optional chaining is a feature introduced in ES11 (ECMAScript 2020) that allows you to safely access nested properties of an object without having to explicitly check if each level of nesting exists. It helps prevent errors and unexpected behavior when trying to access properties of an object that may be null or undefined.

Example of optional chaining:

```javascript
const person = {
  name: {
    first: "Alice"
  }
};

const lastName = person.name?.last; // Safe access even if "name" is undefined

console.log(lastName); // Output: undefined
```

In this example, the `?.` operator checks if the `name` property exists before attempting to access its `last` property. If `name` is undefined, `lastName` will also be undefined.

Ternary Operator (`condition ? expr1 : expr2`):

The ternary operator is a conditional operator that provides a concise way to write conditional statements. It allows you to choose between two expressions based on a condition. It's often used for making decisions and returning different values depending on the condition's evaluation.

Example of the ternary operator:

```javascript
const age = 25;

const message = age >= 18 ? "You are an adult" : "You are a minor";

console.log(message); // Output: You are an adult
```

In this example, the ternary operator checks if `age` is greater than or equal to 18. If the condition is true, it evaluates to `"You are an adult"`, otherwise, it evaluates to `"You are a minor"`.

Differences:

1. **Purpose:**
   - Optional chaining is used for safe access to nested properties, preventing errors when accessing properties of potentially null or undefined objects.
   - The ternary operator is used for conditional expressions, making decisions and choosing between two different values or expressions based on a condition.

2. **Use Case:**
   - Use optional chaining when accessing properties of objects that might not exist at every level of nesting.

- Use the ternary operator when you need to perform a conditional operation and choose between two values or expressions based on a condition.

In summary, optional chaining and the ternary operator have distinct purposes. Optional chaining is used for safe property access, while the ternary operator is used for conditional expressions and decision-making.

9. What do you mean by dot notation and bracket notation? When should you use dot notation or bracket notation?

Answer: **Dot Notation:**

Dot notation is a way to access properties and methods of objects using the dot (`.`) operator followed by the property or method name. It is a concise and commonly used syntax for accessing object members.

Example of dot notation:

```javascript
const person = {
  firstName: "Alice",
  lastName: "Smith",
};

console.log(person.firstName); // Output: Alice
```

Bracket Notation:

Bracket notation is an alternative way to access properties and methods of objects using square brackets `[]`. It allows you to access properties using a string that specifies the property name.

Example of bracket notation:

```javascript
```

```
const person = {
  firstName: "Alice",
  lastName: "Smith",
};

console.log(person["firstName"]); // Output: Alice
```

When to Use Dot Notation:

Use dot notation when:

- The property name is a valid identifier (starts with a letter, doesn't contain spaces or special characters except `_`).

- The property name is known at development time and can be directly written in the code.

When to Use Bracket Notation:

Use bracket notation when:

- The property name is dynamic and is stored in a variable or generated at runtime.

- The property name contains special characters that are not valid identifiers.

- The property name is an expression (e.g., containing spaces or other non-alphanumeric characters).

Examples of using bracket notation:

```javascript
const person = {
  firstName: "Alice",
  lastName: "Smith",
};

const prop = "firstName";
```

```
console.log(person[prop]); // Output: Alice

const specialProp = "last-name";
console.log(person[specialProp]); // Output: Smith
```

Summary:

- Dot notation is used when the property name is a valid identifier and known at development time.
- Bracket notation is used when the property name is dynamic, an expression, or contains special characters.
- Both notations serve the same purpose and are interchangeable in most cases. The choice depends on the context and the specific needs of your code.

Remember that dot notation is often preferred for its readability and simplicity, while bracket notation is more versatile when dealing with dynamic property names or special characters.