Javascript Built-In Methods

Referat about JavaScript Built-In Methods

Understanding and Using JavaScript Built-in Methods Built-in methods are available for various data types, including arrays, strings, numbers, symbols, bigInts, bools and objects. One of the biggest advantages of using built-in methods is that they are highly optimized, so they are often faster and more efficient than writing custom code. In this Referat, we will learn basics and explore the various built-in methods available in JavaScript, including their syntax, usage, and practical examples.

Number Methods

1. The Number.isFinite() method determines the passed number is an finite or not.

It takes one argument 'any integer' and returns 'true' or 'false'. Here are some examples.

```
Number.isFinite(Infinity); // false
Number.isFinite(NaN); // false
Number.isFinite(-Infinity); // false

Number.isFinite(0); // true
Number.isFinite(2e64); // true
```

Javascript also have an isFinite() global function. Difference between Number.isFinite() and isFinite() is type conversion. Number.isFinite() only works with integer numbers not strings. We can see that in following example

```
Number.isFinite('0'); // false because '0' is a string not an number
isFinite('0'); // true
```

2. The Number.isInteger() static method determines whether the passed value is an integer.

It takes one argument 'any value' and returns 'true' or 'false'. Here are some examples.

```
Number.isInteger(0); // true
Number.isInteger(1); // true
Number.isInteger(-100000); // true
Number.isInteger(99999999999999999999); // true

Number.isInteger(0.1); // false
Number.isInteger(Math.PI); // false

Number.isInteger(NaN); // false
```

```
Number.isInteger(Infinity); // false
Number.isInteger(-Infinity); // false
Number.isInteger("10"); // false
Number.isInteger(true); // false
Number.isInteger(false); // false
Number.isInteger([1]); // false
```

Note that some number literals, while looking like non-integers, actually represent integers — due to the precision limit of ECMAScript floating-point number encoding (IEEE-754). For example.

```
Number.isInteger(5.0); // true
Number.isInteger(5.000000000000001); // false
Number.isInteger(5.0000000000000001); // true, because of loss of precision
Number.isInteger(4500000000000000.1); // true, because of loss of precision
```

3. The Number.isNaN() static method determines whether the passed value is the number value NaN, and returns false if the input is not of the Number type.

Here are some examples.

```
Number.isNaN(NaN); // true
Number.isNaN(Number.NaN); // true
Number.isNaN(0 / 0); // true
Number.isNaN(37); // false
```

Javascript also have an isNaN() global function.The difference between Number.isNan() and isNaN() is type conversion.The Number.isNaN() only works with integer type values,if argument is an string, the method will always return false. **Number.isNaN() method**

```
Number.isNaN("NaN"); // false => string
Number.isNaN(undefined); // false => undefined
Number.isNaN({}); // false => object
Number.isNaN("blabla"); // false => string
```

isNaN() global function

```
isNaN("NaN"); // true => NaN == NaN
isNaN(undefined); // true =>
isNaN({}); // true
isNaN("blabla"); // true
```

4. The Number.isSafeInteger() static method determines whether the provided value is a number that is a safe integer.

The safe integers consist of all integers from -(253 - 1) to 253 - 1, inclusive (±9,007,199,254,740,991).Here are some examples

```
Number.isSafeInteger(3); // true
Number.isSafeInteger(2 ** 53); // false
Number.isSafeInteger(2 ** 53 - 1); // true
Number.isSafeInteger(NaN); // false
Number.isSafeInteger(Infinity); // false
Number.isSafeInteger("3"); // false
Number.isSafeInteger(3.1); // false
Number.isSafeInteger(3.0); // true
```

5. The Number.parseFloat() static method parses an argument and returns a floating point number.

If a number cannot be parsed from the argument, it returns NaN. It takes as an argument string.

```
Number.parseFloat("123.1"); // 123.1
Number.parseFloat("123"); // 123
Number.parseFloat("12.345e6"); // 1234500
Number.parseFloat("hello world"); // NaN
```

6. The Number.parseInt() method in JavaScript is used to convert a string representation of a number to an integer.

Here's the syntax for the Number.parseInt() method:

```
Number.parseInt(string [, radix])
```

where string is the string representation of the number you want to convert, and radix is an optional argument specifying the base of the number to be converted. The default value of radix is 10.

Here are a few examples to demonstrate how the Number.parseInt() method can be used:

```
Number.parseInt("123"); // 123
Number.parseInt("1011", 2); // 11
Number.parseInt("14", 8); // 12
Number.parseInt("JavaScript"); // NaN
```

7. The toExponential() method returns a string representing the Number object in exponential notation

Here's the syntax for the toExponential() method:

```
toExponential()
toExponential(fractionDigits)
```

A string representing the given Number object in exponential notation with one digit before the decimal point, rounded to fractionDigits digits after the decimal point.

```
const numObj = 77.1234;

console.log(numObj.toExponential()); // 7.71234e+1
console.log(numObj.toExponential(4)); // 7.7123e+1
console.log(numObj.toExponential(2)); // 7.71e+1
console.log(77.1234.toExponential()); // 7.71234e+1
console.log((77).toExponential()); // 7.7e+1
```

8. The toFixed() method formats a number using fixed-point notation.

Here's the syntax

```
toFixed()
toFixed(digits)
```

The toFixed() method returns a string representation of numObj that does not use exponential notation and has exactly digits digits after the decimal place. The number is rounded if necessary, and the fractional part is padded with zeros if necessary so that it has the specified length.

```
const numObj = 12345.6789;

numObj.toFixed(); // '12346'; rounding, no fractional part
numObj.toFixed(1); // '12345.7'; it rounds up
numObj.toFixed(6); // '12345.678900'; additional zeros
(1.23e+20).toFixed(2); // '123000000000000000000.00'
(1.23e-10).toFixed(2); // '0.00'
2.34.toFixed(1); // '2.3'
2.35.toFixed(1); // '2.4'; it rounds up
2.55.toFixed(1); // '2.5'
// it rounds down as it can't be represented exactly by a float and the
// closest representable float is lower
2.449999999999999.toFixed(1); // '2.5'
// it rounds up as it's less than NUMBER.EPSILON away from 2.45.
// This literal actually encodes the same number value as 2.45

(6.02 * 10 ** 23).toFixed(50); // 6.019999999999999e+23; large numbers still use exponentia
```

9. The toLocaleString() method returns a string with a language-sensitive representation of this number.

Here's the syntax

```
toLocaleString()
toLocaleString(locales)
toLocaleString(locales, options)
```

The locales and options parameters customize the behavior of the function and let applications specify the language whose formatting conventions should be used.

```
const number = 3500;

console.log(number.toLocaleString()); // "3,500" if in U.S. English locale
```

We can see another example to understand method toLocaleString()

```
number.toLocaleString('ar-EG'); // ٣٥٠٠
```

It outputs 3500 in Arabic language ### 10. The toPrecision() method in JavaScript is used to format a number by specifying the number of significant digits to display. Here's an example of using the toPrecision() method:

```
(123.456).toPrecision(2); //  "1.2e+2"
(123.456).toPrecision(5); //  "123.46"
```

In this example, the toPrecision() method is called on a number and is passed a number of significant digits to display. The method returns a string representation of the number, formatted with the specified number of significant digits. The string may include scientific notation, depending on the number and the number of significant digits specified. ### 11. The toString() method in JavaScript is a method that is used to convert a number to a string. The general syntax for using the toString() method is:

```
number.toString([radix]);
```

where number is a number value and radix is an optional parameter that specifies the base to use for representing the number as a string. The radix can be an integer between 2 and 36. If radix is not specified, it defaults to 10.

Here are some examples of using the toString() method in JavaScript:

```
(123).toString()); // "123"
(123).toString(2); // "1111011"
(123).toString(16); // "7b"
```

12. The valueOf() method in JavaScript is a method that is inherent to all JavaScript objects.

It returns the primitive value of the object. For most objects, the default implementation of the valueOf() method returns the object itself, which is not very useful. However, for certain built-in objects, such as Number, Boolean, and Date, the valueOf() method returns a primitive value that represents the object.

For example:

```
let num = 123;
num.valueOf(); // 123
```

```
let bool = true;
bool.valueOf(); // true

let date = new Date();
date.valueOf(); // the number of milliseconds since January 1, 1970, 00:00:00 UTC
```

Boolean Methods

In JavaScript, there are several built-in methods that return a Boolean value (true or false). Some of the most commonly used ones are ### 1. The Array.includes() returns true if an array includes a certain element, and false otherwise.

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

// Check if an array includes a certain element Example 1
fruits.includes('apple');  // true
fruits.includes('pear');   // false


// Check if an array includes an element with a specific index Example 2
fruits.includes('banana', 1);   // true
fruits.includes('banana', 2);   // false
```

In the example, we create an array of fruits and use the includes() method to check if the array includes the elements 'apple' and 'pear'. In the second example, we also use the includes() method but with a second argument that specifies the index at which to start the search.

2. The includes() method in JavaScript is used to check if a given string contains a specified substring or not.

It returns a boolean value (true if the substring is present, false otherwise). Here are some examples to demonstrate its usage:

```
let str = "Hello, World!";

// Check if the string contains the substring "Hello"
console.log(str.includes("Hello")); // true

// Check if the string contains the substring "Goodbye"
console.log(str.includes("Goodbye")); // false

// Check if the string contains the substring "World" starting from index 7
console.log(str.includes("World", 7)); // true
```

```
// Check if the string contains the substring "world" (case sensitive)
console.log(str.includes("world")); // false
```

You can also use the includes() method with arrays to check if an array contains a specified element. For example:

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

// Check if the array contains the element 3
console.log(arr.includes(3)); // true

// Check if the array contains the element 6
console.log(arr.includes(6)); // false
```

String Methods

1. The charAt() method in JavaScript is used to get the character at a specified index in a string.

Here's an example to demonstrate its usage:

```
let str = "Hello World!";

str.charAt(0); // H
```

The square bracket notation and this Method is equivalent

```
str[0]; // H
```

2. The charCodeAt() method in JavaScript is used to get the Unicode code of the character at a specified index in a string.

The Unicode code is a numerical representation of a character, and it ranges from 0 to 65535.

```
let str = "Hello, World!";

// Get the Unicode code of the character at index 0
str.charCodeAt(0); // 72

// Get the Unicode code of the character at index 7
str.charCodeAt(7); // 87

// Get the Unicode code of the character at index 100 (beyond the length of the string)
str.charCodeAt(100); // NaN
```

3. The concat() method in JavaScript is used to concatenate two or more strings together into a single string.

Here's an example to demonstrate its usage:

```
let str1 = "Hello";
let str2 = "Methods";

// Concatenate str1 and str2
let result = str1.concat(" ", str2, "!");
result // "Hello Methods!"
```

We can also concatenate strings using the + operator

```
let str1 = "Hello";
let str2 = "Methods";

// Concatenate str1 and str2
let result = str1 + " " + str2 + "!";
result; // "Hello Methods!"
```

4. The indexOf() method in JavaScript is used to search for a specified substring in a string, and returns the index of the first occurrence of the substring.

If the substring is not found, indexOf() returns -1. Here's an example to demonstrate its usage:

```
let str = "Hello World!";

str.indexOf("Hello"); // 0
str.indexOf("world"); // -1
```

5. The lastIndexOf() method in JavaScript is used to search for a specified substring in a string, starting from the end of the string, and returns the index of the last occurrence of the substring.

If the substring is not found, lastIndexOf() returns -1. Here's an example to demonstrate its usage:

```
let str = "developer";

str.lastIndexOf("e"); // 7
```

6. The localeCompare() method in JavaScript is used to compare two strings in the current locale and returns a number indicating their relative order.

The returned value is:

Less than 0 if the first string comes before the second string in the current locale's sort order. 0 if the two strings are equal in the current locale's sort order. Greater than 0 if the first string comes after the second string in the current locale's sort order. Here's an example to demonstrate its usage:

```
let str1 = "ä";
let str2 = "z";

// Compare the two strings in the current locale
console.log(str1.localeCompare(str2)); // -1

let str3 = "a";
let str4 = "z";

// Compare the two strings in the current locale
console.log(str3.localeCompare(str4)); // -1
```

In the example above, localeCompare(str2) returns -1, indicating that the first string “ä” comes before the second string “z” in the current locale’s sort order. Similarly, localeCompare(str4) returns -1, indicating that the first string “a” comes before the second string “z” in the current locale’s sort order. ### 7. The length property in JavaScript is used to determine the number of elements in an object. This property is commonly used with arrays and strings.

For example, you can use the length property to find the number of elements in an array:

```
let arr = ["push","pop","shift","unshift"];

arr.length; // 4
```

We can delete last element using length property

```
arr.length = arr.length - 1;
arr.length; // 3
arr // ["push","pop","shift"]
```

If we call length in string it will output the count of characters

```
let name = "John Doe";
name.length; // 8
```

8. The match() method in JavaScript is used to search a string for a match against a regular expression and returns the matching result as an array.

Here’s an example of using the match() method to search for a pattern in a string:

```
let str = "Understanding and Using Built In Javascript methods";
let pattern = /[A-Z]/g;
let matches = str.match(pattern);
matches // ['U','U','B','I','J']
```

In this example, the match() method searches the string str for any uppercase

letters and returns the matching characters in an array. The /[A-Z]/g regular expression specifies that we want to match uppercase letters, and the g flag indicates that we want to search for multiple occurrences of the pattern in the string. ***If there are no matches, the match() method returns null.***

```
str.match(/[1-10]/g) // null
```

In JavaScript, flags are used in regular expressions to modify the behavior of the match() method and other regular expression methods. 1) g (global) flag: When the g flag is used, the match() method will return all matches in the string, not just the first one. For example:

```
let str = "The quick brown fox jumps over the lazy dog.";
let pattern = /the/gi;
let matches = str.match(pattern);
matches; // Output: ["The", "the"]
```

2) i (ignore case) flag: When the i flag is used, the match() method will perform a case-insensitive match. For example:

```
let str = "The quick brown FOX jumps over the lazy dog.";
let pattern = /fox/i;
let matches = str.match(pattern);
matches; // ["FOX"]
```

3) m (multiline) flag: When the m flag is used, the ^ and $ symbols in the regular expression will match the beginning and end of each line, respectively, in a multiline string. For example:

```
let str = "The quick\nbrown fox\njumps over\nthe lazy dog.";
let pattern = /^the/gm;
let matches = str.match(pattern);
matches; // Output: ["The", "the"]
```

9. The replace() method in JavaScript is used to search and replace a string with another string.

It can be used to change all occurrences of a specific pattern in a string, or just the first occurrence.

Here's an example of using the replace() method to replace all occurrences of a pattern in a string:

```
let str = "Understanding and Using Built in functions";
let pattern = /functions/gi;
let newStr = str.replace(pattern, "methods");
newStr; //  "Understanding and Using Built in methods"
```

In this example, the replace() method searches the string str for all occurrences of the pattern the, and replaces them with the string "a". The /the/gi regular expression specifies that we want to match the string "the" in a case-insensitive

manner, and the g flag indicates that we want to replace all occurrences of the pattern in the string. ### 10. The search() method in JavaScript is used to search for a specified pattern in a string and returns the position of the first occurrence of the pattern.

Here's an example of using the search() method to find the first occurrence of a pattern in a string:

```
let str = "The quick brown fox jumps over the lazy dog.";
let pattern = /the/i;
let position = str.search(pattern);
console.log(position); // Output: 0
```

In this example, the search() method searches the string str for the first occurrence of the pattern the. The /the/i regular expression specifies that we want to match the string "the" in a case-insensitive manner.

The search() method returns the position of the first occurrence of the pattern in the string. If the pattern is not found, the method returns -1.

It's worth noting that the search() method is similar to the indexOf() method in JavaScript, but the search() method takes a regular expression as its argument, whereas the indexOf() method takes a string.

11. The slice() method in JavaScript is used to extract a part of an array or a string and return a new array or a string, respectively.

Here's an example of using the slice() method with an array:

```
const fruits = ['apple', 'banana', 'kiwi', 'mango'];

const slicedFruits = fruits.slice(1, 3);
slicedFruits; // ['banana', 'kiwi']
```

In this example, the slice() method is used to extract elements from the fruits array starting from the index 1 (inclusive) and ending at index 3 (exclusive), and return a new array slicedFruits containing the extracted elements ['banana', 'kiwi'].

Here's an example of using the slice() method with a string:

```
const sentence = "Hello, World!";

const slicedSentence = sentence.slice(7, 12);
slicedSentence; // 'World'
```

In this example, the slice() method is used to extract characters from the sentence string starting from the index 7 (inclusive) and ending at index 12 (exclusive), and return a new string slicedSentence containing the extracted characters 'World'. ### 12. The split() method in JavaScript is used to split a string

into an array of substrings based on a specified separator. Here's an example of using the split() method:

```
let sentence = "Hello, World!";
let words = sentence.split(" ");
words; // ['Hello,', 'World!']
```

Here's another example of using the split() method with a different separator:

```
let birthday = "17-03-2005";
let birtharr = birthday.split("-");
birtharr; // ['17','03','2005']
```

13. The substr() method in JavaScript is used to extract a part of a string and return a new string.

Here's an example of using the substr() method:

```
const sentence = "Hello, World!";

const subSentence = sentence.substr(7, 5);
subSentence; // 'World'
```

In this example, the substr() method is used to extract a part of the sentence string starting from the index 7 and containing 5 characters. The method returns a new string subSentence containing the extracted characters 'World'.

```
const message = "Good morning!";

const subMessage = message.substr(5, 8);
subMessage; // 'morning'
```

In this example, the substr() method is used to extract a part of the message string starting from the index 5 and containing 8 characters. The method returns a new string subMessage containing the extracted characters 'morning'.

Javascript also have a ***substring()*** method.There are some differences between the two methods: 1. Starting and ending indices: The substr() method takes two arguments: the starting index and the length of the substring to extract. The substring() method takes two arguments: the starting index and the ending index (exclusive).

2. Negative arguments: If either argument to substr() is negative, it is treated as zero. If either argument to substring() is negative, it is treated as the length of the string plus the argument (i.e. a negative argument counts from the end of the string).

Here's an example to demonstrate the difference:

```
const message = "Good morning!";
```

```
const substrMessage = message.substr(5, 8);
const substringMessage = message.substring(5, 13);
console.log(substrMessage); // 'morning'
console.log(substringMessage); // 'morning'
```

In this example, both the substr() and substring() methods extract a part of the message string. The substr() method takes two arguments: 5 (the starting index) and 8 (the length of the substring to extract). The method returns the 'morning' substring. The substring() method takes two arguments: 5 (the starting index) and 13 (the ending index). The method also returns the 'morning' substring. If we change the ending index in the substring() method to a number greater than the length of the string, it will automatically adjust the ending index to the length of the string:

```
const substringMessage = message.substring(5, 20);
console.log(substringMessage); // 'morning!'
```

In this example, the substring() method takes two arguments: 5 (the starting index) and 20 (the ending index). Since the string message has a length of only 12, the ending index is automatically adjusted to 12, so the method returns the substring 'morning!'. ### 14. The toLocaleLowerCase() method in JavaScript is used to convert a string to lowercase based on the locale. Here are some examples of how to use this method:

```
let msg = "Javascript Methods";
let lower_msg = msg.toLocaleLowerCase();
lower_msg // "javascript methods"
```

In this example, the toLocaleLowerCase() method is used to convert the message string to lowercase. The resulting string is "javascript methods".

```
let msg = "HÄLLO WÖRLD";
let lower_msg = msg.toLocaleLowerCase();
lower_msg // "hällo wörld"
```

In this example, the toLocaleLowerCase() method is used to convert a string with uppercase characters with diacritical marks. The resulting string is 'hällo wörld'.

It's important to note that the behavior of toLocaleLowerCase() may vary based on the locale and the platform. ### 15. The toLocaleUpperCase() method in JavaScript is used to convert a string to uppercase based on the locale. Here are some examples of how to use this method:

```
const message = "Hello World";
const upperCaseMessage = message.toLocaleUpperCase();
upperCaseMessage; // 'HELLO WORLD'
```

toLocaleUpperCase() method like an toLocaleLowerCase() have vary behavior based on the locale and the platform. ### 16. The toLowerCase() method in

JavaScript is used to convert a string to lowercase. Here's an example of how to use this method:

```
const message = "Hello World";
const lowerCaseMessage = message.toLowerCase();
lowerCaseMessage; // 'hello world'
```

In this example, the toLowerCase() method is used to convert the message string to lowercase. The resulting string is 'hello world'.

Note that message string is not changing. The method toLowerCase() creates new string. Because string is immutable type. ### 17. The toUpperCase() method in JavaScript is used to convert a string to uppercase. Here's an example of how to use this method:

```
const message = "Hello World";
const upperCaseMessage = message.toUpperCase();
upperCaseMessage; // 'HELLO WORLD'
```

In this example, the toUpperCase() method is used to convert the message string to uppercase. The resulting string is 'HELLO WORLD'.

Note that message string is not changing. The method toUpperCase() creates new string. Because string is immutable type. ### 18. The toString() method in JavaScript is used to convert a value to a string. Here are some examples of using the toString() method:

```
(123).toString(); // '123'
(true).toString(); // 'true'
```

In addition to its basic use for converting values to strings, the toString() method also has some additional capabilities. For example, when used with numbers, it can be used to convert a number to a binary, octal, or hexadecimal string representation:

```
(10).toString(2); // '1010'
(10).toString(8); // '12'
(10).toString(16).toUpperCase(); // 'A'
```

In these examples, the toString() method is passed a radix (the base of the number system), and the resulting string representation is in binary, octal, or hexadecimal, respectively. ### 19. The endsWith() method in JavaScript checks if a string ends with the specified characters. Here's an syntax of endsWith()

```
endsWith(searchString)
endsWith(searchString, endPosition)
```

Here's an example of how you can use it:

```
let str = "Hello, World!";
let endsWithWorld = str.endsWith("World!");
```

```
endsWithWorld; // true
```

In this example, the endsWith() method is used to check if the string str ends with the characters "World!". The result of this method will be a Boolean value that indicates whether the string ends with the specified characters or not. In this case, endsWithWorld will be true, since str does end with "World!".

You can also specify the starting index for the search using the second parameter of the endsWith() method:

```
let str = "Hello, World!";
let endsWithHello = str.endsWith("Hello", 5);

endsWithHello; // false
```

In this example, the endsWith() method is used to check if the string str ends with the characters "Hello" starting from the index 5. The result of this method will be false, since the characters "Hello" do not appear at the end of the string when starting the search from the index 5. ### 20. The String.fromCharCode() method in JavaScript allows you to convert a sequence of Unicode code points into a string. Here's an example of how you can use it:

```
let string = String.fromCharCode(74,97,118,97);
string; // "Java"
```

In this example, the String.fromCharCode() method is used to convert the Unicode code points 74, 97, 118 and 97 into a string. The result will be the string "Java".

You can also use the String.fromCharCode() method to convert multiple code points into a string:

```
let string = String.fromCharCode(74,97,118,97,115,99,114,105,112,116);
string; // "Javascript"
```

In this example, the String.fromCharCode() method is used to convert the sequence of code points into the string "Javascript".

Note that the String.fromCharCode() method is a static method, meaning that you call it directly on the String object, rather than on an instance of a string. ### 21. The String.fromCodePoint() method in JavaScript is similar to String.fromCharCode() but it can handle a wider range of Unicode code points, including those that are greater than 65,535. Here's an example of how you can use it:

```
let string = String.fromCodePoint(74,97,118,97,115,99,114,105,112,116);
string; // "Javascript"
```

In this example, the String.fromCodePoint() method is used to convert the code points 74,97,118,97,115,99,114,105,112 and 116 into a string. The result will be the string "Javascript".

You can also use the String.fromCodePoint() method to convert code points into a string, even if they are greater than 65,535:

```
let string = String.fromCodePoint(128512);

string // " "
```

In this example, the String.fromCodePoint() method is used to convert the code point 128512 into a string. The result will be the string representation of the Unicode character " " (a smiling face with open mouth and tightly-closed eyes).

Note that the String.fromCodePoint() method is a static method, meaning that you call it directly on the String object, rather than on an instance of a string. ### 22. The indexOf() method in JavaScript is used to find the first occurrence of a specified value in a string. Here's an example of how you can use it:

```
let str = "Hello, World!";
let indexOfWorld = str.indexOf("World");

indexOfWorld; // 7
```

In this example, the indexOf() method is used to find the first occurrence of the string "World" in the string str. The result of this method will be the index of the first occurrence of the specified value in the string, or -1 if the value is not found. In this case, indexOfWorld will be 7, since the first occurrence of "World" in str starts at index 7.

You can also specify the starting index for the search using the second parameter of the indexOf() method:

```
let str = "Hello, World!";
let indexOfHello = str.indexOf("Hello", 5);
indexOfHello; // -1
```

In this example, the indexOf() method is used to find the first occurrence of the string "Hello" in the string str starting from the index 5. The result of this method will be -1, since the characters "Hello" do not appear in the string str when starting the search from the index 5. ### 23. The lastIndexOf() method in JavaScript is used to find the last occurrence of a specified value in a string. Here's an example of how you can use it:

```
let str = "Hello, World! Hello, World!";
let lastIndexOfHello = str.lastIndexOf("Hello");
lastIndexOfHello; // 13
```

In this example, the lastIndexOf() method is used to find the last occurrence of the string "Hello" in the string str. The result of this method will be the index of the last occurrence of the specified value in the string, or -1 if the value is not found. In this case, lastIndexOfHello will be 13, since the last occurrence of "Hello" in str starts at index 13.

You can also specify the starting index for the search using the second parameter of the lastIndexOf() method:

```
let str = "Hello, World! Hello, World!";
let lastIndexOfHello = str.lastIndexOf("Hello", 5);
lastIndexOfHello; // 0
```

In this example, the lastIndexOf() method is used to find the last occurrence of the string "Hello" in the string str starting from the index 5. The result of this method will be 0, since the characters "Hello" first appear in the string str starting from the index 0, and when starting the search from the index 5, it finds the first occurrence of the string "Hello". ### 24. The matchAll() method in JavaScript is used to find all occurrences of a specified regular expression in a string. This method returns an iterator that provides access to the individual matches of the regular expression in the string. Here's an example of how you can use it:

```
let str = "Hello, World! Hello, World!";
let regex = /l/g;

let matches = str.matchAll(regex);

for (let match of matches) {
    console.log(match[0]); 'l' appears 6 times
}
```

In this example, the matchAll() method is used to find all occurrences of the regular expression /Hello/g in the string str. The result of this method will be an iterator that provides access to the individual matches of the regular expression in the string. In this case, the loop will log "l" 6 times, since "l" appears 6 times in the string str.

Note that the matchAll() method is a relatively new addition to the JavaScript language and may not be supported by older browsers. It's also worth noting that this method requires the g (global) flag to be set on the regular expression in order to find all occurrences of the pattern in the string. ### 25. The normalize() method in JavaScript is used to normalize a string representation of a Unicode character sequence, such that it can be compared in a case-insensitive manner. This method returns a new string that represents the same character sequence as the original string, but with the characters in a standardized form that is appropriate for comparison. Here's an example of how you can use it:

```
let str = "m\u00e9dical";
let normalized = str.normalize();

console.log(normalized); // "médical"
```

In this example, the normalize() method is used to normalize the string str, which contains the character é (U+00E9) encoded as a combining character

sequence. The result of this method will be a new string that represents the same character sequence as the original string, but with the characters in a standardized form. In this case, the value of normalized will be “médical”.

You can also specify the normalization form to be used by passing an optional argument to the normalize() method. For example, you can use normalize(“NFD”) to normalize the string using the Normalization Form Canonical Decomposition (NFD) algorithm, or normalize(“NFC”) to normalize the string using the Normalization Form Canonical Composition (NFC) algorithm. ### 26. The padEnd() method in JavaScript is used to pad a string with a specified character or set of characters so that the string reaches a specified length. Here’s an example of how you can use it:

```
let str = "Hello";
let padded = str.padEnd(10, "!");

console.log(padded); //  "Hello!!!!!!".
```

In this example, the padEnd() method is used to pad the string str with the character ! so that the resulting string reaches a length of 10 characters. The result of this method will be a new string that represents the original string with the specified padding characters added to the end. In this case, the value of padded will be “Hello!!!!!!”.

If the original string already exceeds the specified length, then the padEnd() method will not add any padding characters, and the original string will be returned unchanged. If no padding character is specified, then the default padding character is the space character (” “). ### 27. The padStart() method in JavaScript is used to pad a string with a specified character or set of characters so that the string reaches a specified length. This method adds the padding characters to the start of the string, rather than the end as in the case of padEnd(). Here’s an example of how you can use it:

```
let str = "Hello";
let padded = str.padStart(10, "!");

console.log(padded); // "!!!!!Hello"
```

In this example, the padStart() method is used to pad the string str with the character ! so that the resulting string reaches a length of 10 characters. The result of this method will be a new string that represents the original string with the specified padding characters added to the start. In this case, the value of padded will be “!!!!!Hello”.

If the original string already exceeds the specified length, then the padStart() method will not add any padding characters, and the original string will be returned unchanged. If no padding character is specified, then the default padding character is the space character (” “). ### 28. The repeat() method in JavaScript is used to repeat a string a specified number of times to create a new

string. Here's an example of how you can use it:

```
let str = "Hello";
let repeated = str.repeat(3);

console.log(repeated); // "HelloHelloHello"
```

In this example, the repeat() method is used to repeat the string str three times to create a new string. The result of this method will be a new string that consists of the original string repeated the specified number of times. In this case, the value of repeated will be "HelloHelloHello".

The repeat() method will accept a non-negative integer as an argument, and if the argument is not a positive integer, it will return an empty string. If the argument is zero, the original string will be returned unchanged. ### 29. The trim() method in JavaScript is used to remove whitespace characters from the beginning and end of a string. Here's an example of how you can use it:

```
let str = "   Hello World!   ";
let trimmed = str.trim();

console.log(trimmed); "Hello World!"
```

In this example, the trim() method is used to remove any whitespace characters from the beginning and end of the string str. The result of this method will be a new string that has all leading and trailing whitespace characters removed. In this case, the value of trimmed will be "Hello World!".

The trim() method removes all leading and trailing whitespace characters, including spaces, tabs, and line breaks. If you need to remove whitespace characters only from one side of the string, you can use the trimStart() method to remove only leading whitespace characters, or the trimEnd() method to remove only trailing whitespace characters. ### 30. The trimEnd() method in JavaScript is used to remove whitespace characters from the end of a string. Here's an example of how you can use it:

```
let str = "   Hello World!   ";
let trimmed = str.trimEnd();

console.log(trimmed); // " Hello World!"
```

In this example, the trimEnd() method is used to remove any whitespace characters from the end of the string str. The result of this method will be a new string that has all trailing whitespace characters removed. In this case, the value of trimmed will be " Hello World!".

The trimEnd() method removes all trailing whitespace characters, including spaces, tabs, and line breaks. If you need to remove whitespace characters only

from the start of the string, you can use the trimStart() method. If you need to remove whitespace characters from both the start and end of the string, you can use the trim() method. ### 31. The trimStart() method in JavaScript is used to remove whitespace characters from the beginning of a string. Here's an example of how you can use it:

```
let str = "   Hello World!   ";
let trimmed = str.trimStart();

console.log(trimmed); // "Hello World! "
```

In this example, the trimStart() method is used to remove any whitespace characters from the beginning of the string str. The result of this method will be a new string that has all leading whitespace characters removed. In this case, the value of trimmed will be "Hello World!".

The trimStart() method removes all leading whitespace characters, including spaces, tabs, and line breaks. If you need to remove whitespace characters only from the end of the string, you can use the trimEnd() method. If you need to remove whitespace characters from both the start and end of the string, you can use the trim() method. ### ***Array Methods*** ### 1. The Array.prototype.at() method does not exist in JavaScript. JavaScript does have an Array.prototype.indexOf() method that you can use to find the index of an element in an array, but this method returns -1 if the element is not found in the array.

```
let arr = [1, 2, 3, 4, 5];
let element = arr.at(-1);
console.log(element); // Output: 5
```

If you're looking for a way to access an element at a specific index in an array, you can simply use square bracket notation, like this:

```
let arr = [1, 2, 3, 4, 5];
let element = arr[4];
console.log(element); // Output: 3
```

2. The Array.prototype.concat() method in JavaScript is used to merge two or more arrays into a single array.

The method does not modify the original arrays but instead returns a new array that contains the elements from the original arrays.

Here's an example of how you can use Array.prototype.concat():

```
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let newArray = arr1.concat(arr2);
console.log(newArray); // Output: [1, 2, 3, 4, 5, 6]
```

You can also concatenate more than two arrays by passing multiple arrays as arguments to concat(). For example:

```
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let arr3 = [7, 8, 9];
let newArray = arr1.concat(arr2, arr3);
console.log(newArray); // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3. The Array.prototype.copyWithin() method in JavaScript is used to copy a sequence of elements within an array to another location in the same array.

The method modifies the original array and does not create a new array.

Here's the basic syntax for using Array.prototype.copyWithin():

```
array.copyWithin(target, start[, end]);
```

target: The index at which to copy the elements to. start: The index at which to start copying elements from. end: (optional) The index at which to end copying elements (defaults to array.length).

Here's an example of how you can use Array.prototype.copyWithin():

```
let arr = [1, 2, 3, 4, 5];
arr.copyWithin(0, 3);
console.log(arr); // Output: [4, 5, 3, 4, 5]
```

In this example, the elements at indices 3 and 4 (4 and 5) are copied to the beginning of the array (indices 0 and 1), overwriting the existing elements. ### 4. The Array.prototype.entries() method in JavaScript returns a new Array Iterator object that contains the key/value pairs for each index in the array. This can be useful for iterating over an array, as it provides both the index and value of each element in the array.

Here's an example of how you can use Array.prototype.entries():

```
let arr = [1, 2, 3, 4, 5];
let iterator = arr.entries();

for (let [index, value] of iterator) {
  console.log(index, value);
}

// Output:
// 0 1
// 1 2
// 2 3
```

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

In this example, the entries() method returns an iterator object that we can use in a for...of loop to iterate over the elements in the array. The loop destructures each iteration into a key/value pair, where the key is the index and the value is the corresponding element in the array. ### 5. The Array.prototype.every() method in JavaScript is used to test whether all elements in an array pass a test specified by a provided function. The method returns a Boolean value indicating whether all elements pass the test.

Here's the basic syntax for using Array.prototype.every():

```
array.every(callback(element[, index[, array]])[, thisArg]);
```

callback: A function to be run for each element in the array. It should return true or false. thisArg: (optional) An object to be used as this when executing the callback function. Here's an example of how you can use Array.prototype.every():

```
let arr = [1, 2, 3, 4, 5];
let result = arr.every(value => value >= 1);
console.log(result); // Output: true
```

In this example, the every() method tests whether all elements in the arr array are greater than or equal to 1. Since all elements in the array pass this test, the method returns true. ### 6. The Array.prototype.fill() method in JavaScript is used to fill all the elements of an array with a static value, from a start index to an end index. The method modifies the original array and does not create a new array.

Here's the basic syntax for using Array.prototype.fill():

```
array.fill(value[, start[, end]]);
```

value: The value to fill the array with. start: (optional) The index at which to start filling the array (defaults to 0). end: (optional) The index at which to end filling the array (defaults to array.length). Here's an example of how you can use Array.prototype.fill():

```
let arr = [1, 2, 3, 4, 5];
arr.fill(0, 1, 3);
console.log(arr); // Output: [1, 0, 0, 4, 5]
```

In this example, the fill() method fills the elements in the arr array with the value 0 from index 1 to index 3 (inclusive). ### 7. The Array.prototype.filter() method in JavaScript is used to create a new array with all elements that pass a test specified by a provided function. The method does not modify the original array.

Here's the basic syntax for using Array.prototype.filter():

```
array.filter(callback(element[, index[, array]])[, thisArg]);
```

callback: A function to be run for each element in the array. It should return true or false. thisArg: (optional) An object to be used as this when executing the callback function. Here's an example of how you can use Array.prototype.filter():

```
let arr = [1, 2, 3, 4, 5];
let filteredArr = arr.filter(value => value % 2 === 0);
console.log(filteredArr); // Output: [2, 4]
```

In this example, the filter() method creates a new array filteredArr containing only the elements in the arr array that are even (i.e., whose remainder when divided by 2 is 0). The method does not modify the original arr array. ### 8. The Array.prototype.find() method in JavaScript is used to return the value of the first element in an array that satisfies a provided testing function. The method returns undefined if no element passes the test.

Here's the basic syntax for using Array.prototype.find():

```
array.find(callback(element[, index[, array]])[, thisArg]);
```

callback: A function to be run for each element in the array. It should return true or false. thisArg: (optional) An object to be used as this when executing the callback function. Here's an example of how you can use Array.prototype.find():

```
let arr = [1, 2, 3, 4, 5];
let result = arr.find(value => value > 3);
console.log(result); // Output: 4
```

In this example, the find() method returns the first value in the arr array that is greater than 3. Since the first such value is 4, the method returns 4. If no element in the array passes the test, the method returns undefined. ### 9. The Array.prototype.findIndex() method in JavaScript is used to return the index of the first element in an array that satisfies a provided testing function. The method returns -1 if no element passes the test.

Here's the basic syntax for using Array.prototype.findIndex():

```
array.findIndex(callback(element[, index[, array]])[, thisArg]);
```

callback: A function to be run for each element in the array. It should return true or false. thisArg: (optional) An object to be used as this when executing the callback function. Here's an example of how you can use Array.prototype.findIndex():

```
let arr = [1, 2, 3, 4, 5];
let result = arr.findIndex(value => value > 3);
console.log(result); // Output: 3
```

In this example, the findIndex() method returns the index of the first value in the arr array that is greater than 3. Since the first such value is 4, the method

returns 3. If no element in the array passes the test, the method returns -1. ### 10. The Array.prototype.findLastIndex() method is a built-in JavaScript method that returns the last index of the last element in the array that satisfies the provided testing function.

Here is the syntax for using Array.prototype.findLastIndex():

```
array.findLastIndex(callback(element[, index[, array]])[, thisArg])
```

callback: A function that is called for each element in the array, taking the following arguments: element, index, and array. The function should return true if the element satisfies the condition and false otherwise. thisArg (optional): Object to use as this when executing callback. The method returns the index of the first element that satisfies the provided testing function, or -1 if none of the elements in the array pass the test.

Here is an example of using Array.prototype.findLastIndex():

```
let arr = [5, 12, 8, 130, 44];

let result = arr.findLastIndex(element => element >= 12);

console.log(result); // 3
```

In this example, the findLastIndex() method returns the index of the last element in the array that is greater than or equal to 12, which is 3. ### 11. The Array.prototype.flat() method is a built-in JavaScript method that creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

Here is the syntax for using Array.prototype.flat():

```
array.flat([depth])
```

depth (optional): The maximum recursion depth. Default value is 1. The method returns a new array with all the sub-array elements concatenated into it. If the depth argument is provided, it will only flatten the array to that depth.

Here is an example of using Array.prototype.flat():

```
let arr = [1, 2, [3, 4, [5, 6]]];

let result = arr.flat();

console.log(result); // [1, 2, 3, 4, [5, 6]]

result = arr.flat(2);

console.log(result); // [1, 2, 3, 4, 5, 6]
```

In this example, the flat() method flattens the array arr to a depth of 1 in the first call, resulting in [1, 2, 3, 4, [5, 6]]. In the second call, it flattens the array to a

depth of 2, resulting in [1, 2, 3, 4, 5, 6]. ### 12. The Array.prototype.flatMap() method is a built-in JavaScript method that maps each element using a mapping function, then flattens the result into a new array. It is essentially a combination of the map() and flat() methods.

Here is the syntax for using Array.prototype.flatMap():

```
array.flatMap(callback(element[, index[, array]])[, thisArg])
```

callback: A function that is called for each element in the array, taking the following arguments: element, index, and array. The function should return an array or a value that will be flattened. thisArg (optional): Object to use as this when executing callback. The method returns a new array with the results of calling a provided function on every element in the calling array.

Here is an example of using Array.prototype.flatMap():

```
let arr = [1, 2, 3, 4];

let result = arr.flatMap(x => [x * 2]);

console.log(result); // [2, 4, 6, 8]
```

In this example, the flatMap() method maps each element in the array arr using the mapping function x => [x * 2], which returns an array of doubled values. The result of the flatMap() method is then a flattened array of these doubled values. ### 13. The Array.prototype.forEach() method is a built-in JavaScript method that executes a provided function once for each array element.

Here is the syntax for using Array.prototype.forEach():

```
array.forEach(callback(element[, index[, array]])[, thisArg])
```

callback: A function that is called for each element in the array, taking the following arguments: element, index, and array. The function should perform the desired operation for each element. thisArg (optional): Object to use as this when executing callback. The method does not return a value.

Here is an example of using Array.prototype.forEach():

```
let arr = [1, 2, 3, 4];

arr.forEach(function(element, index, array) {
  console.log(element, index);
});

// Output:
// 1 0
// 2 1
// 3 2
// 4 3
```

In this example, the forEach() method executes the provided function (element, index, array) => console.log(element, index) for each element in the array arr. The function logs the element and its index to the console. ### 14. The Array.from() method is a built-in JavaScript method that creates a new, shallow-copied Array instance from an array-like or iterable object.

Here is the syntax for using Array.from():

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

arrayLike: An object to convert to an array. mapFn (optional): Map function to call on every element of the array. thisArg (optional): Object to use as this when executing mapFn. The method returns a new Array instance with the elements of arrayLike.

Here is an example of using Array.from():

```
let arrayLike = {0: 'a', 1: 'b', 2: 'c', length: 3};

let result = Array.from(arrayLike);

console.log(result); // ['a', 'b', 'c']
```

In this example, Array.from() is used to convert the arrayLike object to an array. The result of Array.from() is a new array instance with the elements of arrayLike. ### 14. The Array.prototype.includes() method is a built-in JavaScript method that determines whether an array includes a certain value among its entries, returning true or false as appropriate.

Here is the syntax for using Array.prototype.includes():

```
array.includes(valueToFind[, fromIndex])
```

valueToFind: The value to search for. fromIndex (optional): The position in the array at which to start searching for valueToFind. The method returns a boolean value indicating whether the value was found in the array.

Here is an example of using Array.prototype.includes():

```
let arr = [1, 2, 3, 4];

console.log(arr.includes(3)); // true
console.log(arr.includes(5)); // false
```

In this example, the includes() method is used to determine if the value 3 is present in the array arr. The result of includes() is true, indicating that the value was found in the array. The includes() method is also used to determine if the value 5 is present in the array arr. The result of includes() is false, indicating that the value was not found in the array. ### 15. The Array.prototype.indexOf() method is a built-in JavaScript method that returns the first index at which a given element can be found in the array, or -1 if it is not present.

Here is the syntax for using Array.prototype.indexOf():

```
array.indexOf(searchElement[, fromIndex])
```

searchElement: The value to search for. fromIndex (optional): The position in the array at which to start searching for searchElement. The method returns the first index at which searchElement can be found in the array, or -1 if it is not present.

Here is an example of using Array.prototype.indexOf():

```
let arr = [1, 2, 3, 4];

console.log(arr.indexOf(3)); // 2
console.log(arr.indexOf(5)); // -1
```

In this example, the indexOf() method is used to determine the first index at which the value 3 can be found in the array arr. The result of indexOf() is 2, indicating that the value was found at the second position in the array. The indexOf() method is also used to determine the first index at which the value 5 can be found in the array arr. The result of indexOf() is -1, indicating that the value was not found in the array. ### 16. The Array.isArray() method is a built-in JavaScript method that determines whether an object is an Array.

Here is the syntax for using Array.isArray():

```
Array.isArray(obj)
```

obj: The object to be tested. The method returns true if obj is an Array, and false otherwise.

Here is an example of using Array.isArray():

```
let arr = [1, 2, 3, 4];

console.log(Array.isArray(arr)); // true
console.log(Array.isArray({})); // false
```

In this example, the Array.isArray() method is used to determine if arr is an Array. The result of Array.isArray() is true, indicating that arr is indeed an Array. The Array.isArray() method is also used to determine if {} is an Array. The result of Array.isArray() is false, indicating that {} is not an Array. ### 17. The Array.prototype.join() method is a built-in JavaScript method that creates and returns a string by concatenating all the elements of an array with a specified separator.

Here is the syntax for using Array.prototype.join():

```
array.join([separator])
```

separator (optional): The separator to be used between the elements of the array. If this argument is not provided, the elements of the array are separated

by a comma. The method returns a string that is created by concatenating all the elements of the array with the specified separator.

Here is an example of using Array.prototype.join():

```
let arr = [1, 2, 3, 4];

console.log(arr.join()); // "1,2,3,4"
console.log(arr.join("-")); // "1-2-3-4"
```

In this example, the join() method is used to create and return a string by concatenating all the elements of the array arr. The first call to join() returns a string with the elements of arr separated by commas, while the second call to join() returns a string with the elements of arr separated by hyphens. ### 18. The Array.prototype.keys() method is a built-in JavaScript method that returns a new Array Iterator object that contains the keys for each index in the array.

Here is the syntax for using Array.prototype.keys():

```
array.keys()
```

The method returns a new Array Iterator object that contains the keys (indices) for each element in the array.

Here is an example of using Array.prototype.keys():

```
let arr = [1, 2, 3, 4];

let keys = arr.keys();

console.log(keys.next()); // { value: 0, done: false }
console.log(keys.next()); // { value: 1, done: false }
console.log(keys.next()); // { value: 2, done: false }
console.log(keys.next()); // { value: 3, done: false }
console.log(keys.next()); // { value: undefined, done: true }
```

In this example, the keys() method is used to return a new Array Iterator object that contains the keys (indices) for each element in the array arr. The next() method is then called on the Array Iterator object to retrieve the keys (indices) one at a time. The next() method returns an object with two properties: value and done. The value property contains the current key (index), and the done property indicates whether all the keys have been retrieved. When all the keys have been retrieved, the next() method returns an object with value set to undefined and done set to true. ### 19. The Array.prototype.lastIndexOf() method is a built-in JavaScript method that returns the last index at which a given element can be found in an array, or -1 if it is not present. The search starts from the end of the array and goes towards the beginning.

Here is the syntax for using Array.prototype.lastIndexOf():

```
array.lastIndexOf(searchElement[, fromIndex])
```

searchElement: The element to be searched for in the array. fromIndex (optional): The index at which to start searching backwards in the array. If fromIndex is greater than or equal to the length of the array, the entire array will be searched. If fromIndex is negative, it will be treated as array.length + fromIndex where array.length is the length of the array. The default value of fromIndex is array.length - 1. The method returns the last index at which searchElement can be found in the array, or -1 if it is not present.

Here is an example of using Array.prototype.lastIndexOf():

```
let arr = [1, 2, 3, 4, 2, 1];

console.log(arr.lastIndexOf(2)); // 4
console.log(arr.lastIndexOf(2, 3)); // 1
console.log(arr.lastIndexOf(5)); // -1
```

In this example, the lastIndexOf() method is used to search for the element 2 in the array arr. The first call to lastIndexOf() returns 4, which is the last index at which 2 can be found in arr. The second call to lastIndexOf() returns 1, which is the last index at which 2 can be found in arr before the index 3. The third call to lastIndexOf() returns -1, which indicates that the element 5 is not present in arr. ### 20. The Array.prototype.map() method is a built-in JavaScript method that creates a new array with the results of calling a provided function on every element in the calling array.

Here is the syntax for using Array.prototype.map():

```
array.map(callback(element[, index[, array]])[, thisArg])
```

callback: A function that is called for each element in the array, taking the following arguments: element, index, and array. The function should perform the desired operation for each element and return the resulting value. thisArg (optional): Object to use as this when executing callback. The method returns a new array with the results of calling callback on every element in the original array.

Here is an example of using Array.prototype.map():

```
let arr = [1, 2, 3, 4];

let doubleArr = arr.map(function(element) {
  return element * 2;
});

console.log(doubleArr); // [2, 4, 6, 8]
```

In this example, the map() method is used to double each element in the array arr. The provided function (element) => element * 2 is called for each element in arr and the resulting values are collected into a new array doubleArr. The resulting doubleArr is [2, 4, 6, 8], which is the original arr with each element

doubled. ### 21. The Array.of() method is a built-in JavaScript method that creates a new array instance with a variable number of arguments, regardless of number or type of the arguments.

Here is the syntax for using Array.of():

```
Array.of(element0[, element1[, ...[, elementN]]])
```

element0, element1, ..., elementN: Elements to be passed as arguments to the constructor and make up the new array. The method returns a new array instance with the given elements.

Here is an example of using Array.of():

```
let arr = Array.of(1, 2, 3, 4);
console.log(arr); // [1, 2, 3, 4]
```

In this example, the Array.of() method is used to create a new array instance with the elements 1, 2, 3, 4. The resulting arr is [1, 2, 3, 4].

Note that Array.of() is often used instead of the Array constructor because Array.of() guarantees that the resulting array will have the desired number of elements, whereas the Array constructor can have unexpected behavior when passing a single numeric argument. ### 22. The Array.prototype.pop() method is a built-in JavaScript method that removes the last element from an array and returns that element.

Here is the syntax for using Array.prototype.pop():

```
array.pop()
```

The method returns the removed element. If the array is empty, undefined is returned.

Here is an example of using Array.prototype.pop():

```
let arr = [1, 2, 3, 4];

let popped = arr.pop();
console.log(arr); // [1, 2, 3]
console.log(popped); // 4
```

In this example, the pop() method is used to remove the last element from the array arr. The resulting arr is [1, 2, 3] and the removed element, 4, is returned and stored in the variable popped. ### 23. The Array.prototype.push() method is a built-in JavaScript method that adds one or more elements to the end of an array and returns the new length of the array.

Here is the syntax for using Array.prototype.push():

```
array.push(element1[, element2[, ...[, elementX]]])
```

element1, element2, ..., elementX: Elements to be added to the end of the array. The method returns the new length of the array.

Here is an example of using Array.prototype.push():

```
let arr = [1, 2, 3, 4];

let newLength = arr.push(5, 6);
console.log(arr); // [1, 2, 3, 4, 5, 6]
console.log(newLength); // 6
```

In this example, the push() method is used to add the elements 5 and 6 to the end of the array arr. The resulting arr is [1, 2, 3, 4, 5, 6] and the new length of the array, 6, is returned and stored in the variable newLength. ### 24. The Array.prototype.reduce() method is a built-in JavaScript method that applies a function against an accumulator and each element in the array (from left to right) to reduce the array to a single value.

Here is the syntax for using Array.prototype.reduce():

```
array.reduce(callback(accumulator, currentValue[, currentIndex[, array]])[, initialValue])
```

callback: Function to execute on each value in the array, taking four arguments: accumulator: The accumulator accumulates the callback's return values. It is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied (see below). currentValue: The current element being processed in the array. currentIndex (optional): The index of the current element being processed in the array. array (optional): The array reduce() was called upon. initialValue (optional): Object to use as the first argument to the first call of the callback. The method returns the reduced value.

Here is an example of using Array.prototype.reduce():

```
let arr = [1, 2, 3, 4];

let sum = arr.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
console.log(sum); // 10
```

In this example, the reduce() method is used to calculate the sum of the elements in the array arr. The provided function (accumulator, currentValue) => accumulator + currentValue is executed for each element in the array, starting with an accumulator value of 0, to calculate the sum of the elements. The reduced value, 10, is returned and stored in the variable sum. ### 25. The Array.prototype.reduceRight() method is similar to the Array.prototype.reduce() method, with the difference that it processes the elements of the array from right to left.

Here is the syntax for using Array.prototype.reduceRight():

```
array.reduceRight(callback(accumulator, currentValue[, currentIndex[, array]])[, initialValu
```

callback: Function to execute on each value in the array, taking four arguments: accumulator: The accumulator accumulates the callback's return values. It is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied (see below). currentValue: The current element being processed in the array. currentIndex (optional): The index of the current element being processed in the array. array (optional): The array reduceRight() was called upon. initialValue (optional): Object to use as the first argument to the first call of the callback. The method returns the reduced value.

Here is an example of using Array.prototype.reduceRight():

```
let arr = [1, 2, 3, 4];

let sum = arr.reduceRight(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
console.log(sum); // 10
```

In this example, the reduceRight() method is used to calculate the sum of the elements in the array arr, starting from the rightmost element and working towards the left. The provided function (accumulator, currentValue) => accumulator + currentValue is executed for each element in the array, starting with an accumulator value of 0, to calculate the sum of the elements. The reduced value, 10, is returned and stored in the variable sum. ### 26. The Array.prototype.reverse() method is a built-in JavaScript method that reverses the order of the elements in an array.

Here is the syntax for using Array.prototype.reverse():

```
array.reverse()
```

The method returns the reversed array.

Here is an example of using Array.prototype.reverse():

```
let arr = [1, 2, 3, 4];

let reversed = arr.reverse();
console.log(reversed); // [4, 3, 2, 1]
```

In this example, the reverse() method is used to reverse the order of the elements in the array arr. The reversed array [4, 3, 2, 1] is returned and stored in the variable reversed. ### 27. The Array.prototype.shift() method is a built-in JavaScript method that removes the first element from an array and returns the removed element.

Here is the syntax for using Array.prototype.shift():

```
array.shift()
```

The method returns the removed element.

Here is an example of using Array.prototype.shift():

```
let arr = [1, 2, 3, 4];

let first = arr.shift();
console.log(first); // 1
console.log(arr); // [2, 3, 4]
```

In this example, the shift() method is used to remove the first element 1 from the array arr. The removed element 1 is returned and stored in the variable first. The array arr is now [2, 3, 4]. ### 28. The Array.prototype.slice() method is a built-in JavaScript method that returns a shallow copy of a portion of an array. The portion to be copied is specified by a start and an end index.

Here is the syntax for using Array.prototype.slice():

```
array.slice(start, end)
```

start (optional): The starting index of the portion to be copied (inclusive). If start is negative, it is treated as array.length + start where array.length is the length of the array. end (optional): The ending index of the portion to be copied (exclusive). If end is negative, it is treated as array.length + end. If end is not specified, all elements from start to the end of the array are copied.

The method returns the sliced portion of the array as a new array. The original array is not modified.

Here is an example of using Array.prototype.slice():

```
let arr = [1, 2, 3, 4];

let sliced = arr.slice(1, 3);
console.log(sliced); // [2, 3]
console.log(arr); // [1, 2, 3, 4]
```

In this example, the slice() method is used to return a portion of the array arr from index 1 (inclusive) to index 3 (exclusive). The sliced portion [2, 3] is returned and stored in the variable sliced. The original array arr is not modified and remains [1, 2, 3, 4]. ### 29. The Array.prototype.some() method is a built-in JavaScript method that tests whether at least one element in an array passes the test implemented by the provided function.

Here is the syntax for using Array.prototype.some():

```
array.some(callback(element[, index[, array]])[, thisArg])
```

callback: A function that is called for each element in the array, taking the following arguments: element, index, and array. The function should return a boolean value indicating whether the element passed the test. thisArg (optional): Object to use as this when executing callback.

The method returns a boolean value indicating whether at least one element in the array passed the test.

Here is an example of using Array.prototype.some():

```
let arr = [1, 2, 3, 4];

let result = arr.some(function(element, index, array) {
  return element % 2 === 0;
});

console.log(result); // true
```

In this example, the some() method is used to check if at least one element in the array arr is even. The function (element, index, array) => element % 2 === 0 is passed as the callback and it returns true if the element is even. The some() method returns true if at least one element in the array passed the test and false otherwise. In this case, the some() method returns true because the array arr contains at least one even number (2). ### 30. The Array.prototype.sort() method is a built-in JavaScript method that sorts the elements of an array in place and returns the sorted array.

Here is the syntax for using Array.prototype.sort():

```
array.sort([compareFunction])
```

compareFunction (optional): A function that defines the sort order. It should return a negative, zero, or positive value, depending on the arguments, like:

a negative value if a should be sorted lower than b a positive value if a should be sorted higher than b 0 if a and b are equal and their order doesn't matter. If compareFunction is omitted, the array elements are sorted in lexicographic (alphabetical) order.

Here is an example of using Array.prototype.sort():

```
let arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];

arr.sort();
console.log(arr); // [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

In this example, the sort() method is used to sort the elements of the array arr. Since the compareFunction is omitted, the array elements are sorted in lexicographic order. The sort() method sorts the elements of the array in place and returns the sorted array, which is logged to the console. ### 31. The Array.prototype.splice() method is a built-in JavaScript method that changes the contents of an array by adding, removing, and/or replacing elements.

Here is the syntax for using Array.prototype.splice():

```
array.splice(start[, deleteCount[, item1[, item2[, ...]]]])
```

start: An integer that specifies at what position to start changing the array.

deleteCount (optional): An integer that specifies the number of elements to remove.

item1, item2, ... (optional): The elements to add to the array.

The splice() method returns an array containing the deleted elements (if any). If only one element is removed, an array of one element is returned. If no elements are removed, an empty array is returned.

Here is an example of using Array.prototype.splice():

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

let removed = arr.splice(2, 2);
console.log(arr); // [1, 2, 5]
console.log(removed); // [3, 4]
```

In this example, the splice() method is used to remove two elements (3 and 4) from the array arr, starting at index 2. The removed elements are returned as an array, which is assigned to the variable removed, and logged to the console. The modified array is also logged to the console. ### 32. The Array.prototype.toLocaleString() method is a built-in JavaScript method that returns a string representing the elements of an array, using the default locale.

Here is the syntax for using Array.prototype.toLocaleString():

```
array.toLocaleString()
```

The toLocaleString() method returns a string representation of the elements of the array, using the default locale. The elements are separated by a comma (,) by default.

Here is an example of using Array.prototype.toLocaleString():

```
let arr = [1, 2, 3];
let str = arr.toLocaleString();
console.log(str); // "1,2,3"
```

In this example, the toLocaleString() method is used to convert the elements of the array arr to a string, which is assigned to the variable str and logged to the console. ### 33. The Array.prototype.toString() method is a built-in JavaScript method that returns a string representing the elements of an array.

Here is the syntax for using Array.prototype.toString():

```
array.toString()
```

The toString() method returns a string representation of the elements of the array. The elements are separated by a comma (,) by default.

Here is an example of using Array.prototype.toString():

```
let arr = [1, 2, 3];
let str = arr.toString();
console.log(str); // "1,2,3"
```

In this example, the toString() method is used to convert the elements of the array arr to a string, which is assigned to the variable str and logged to the console. ### 34. The Array.prototype.unshift() method is a built-in JavaScript method that adds elements to the beginning of an array and returns the new length of the array.

Here is the syntax for using Array.prototype.unshift():

```
array.unshift(element1[, element2[, ...[, elementX]]])
```

The unshift() method takes one or more elements as arguments and adds them to the beginning of the array.

Here is an example of using Array.prototype.unshift():

```
let arr = [1, 2, 3];
let newLength = arr.unshift(0);
console.log(arr); // [0, 1, 2, 3]
console.log(newLength); // 4
```

In this example, the unshift() method is used to add the element 0 to the beginning of the array arr. The new length of the array is then assigned to the variable newLength and logged to the console. ### 35. The Array.prototype.values() method is a built-in JavaScript method that returns a new Array Iterator object that contains the values for each index in the array. The Array Iterator object can be used in a for...of loop to iterate over the values in the array.

Here is an example of using Array.prototype.values():

```
let arr = [1, 2, 3];
let iterator = arr.values();
for (let value of iterator) {
  console.log(value);
}

// Output:
// 1
// 2
// 3
```

In this example, the values() method is used to get an Array Iterator object for the array arr. The for...of loop is then used to iterate over the values in the array and log them to the console. ### ***Object Methods*** ### 1. Object.assign() is a method in JavaScript that allows you to copy the values of all enumerable properties of one or more source objects to a target object. The method returns the target object and modifies it in place.

Here's an example of how you could use Object.assign():

```
const obj1 = {a: 1, b: 2};
const obj2 = {c: 3, d: 4};
const obj3 = Object.assign({}, obj1, obj2);
console.log(obj3); // {a: 1, b: 2, c: 3, d: 4}
```

In the example above, obj1 and obj2 are source objects, and {} is the target object. The values of obj1 and obj2 are copied to the target object obj3, and the resulting object is {a: 1, b: 2, c: 3, d: 4}.

It's worth noting that Object.assign() only performs a shallow copy of the properties and does not copy properties that are not enumerable (e.g., properties with Symbol keys). Additionally, if the source object has a property with the same key as the target object, the value of the target property will be overwritten by the value of the source property. ### 2. Object.create() is a method in JavaScript that creates an object with the specified prototype object and properties. The Object.create() method allows you to create an object that inherits directly from the passed-in prototype object. This is in contrast to new operator, which creates an object that inherits from the constructor's prototype property.

Here's an example of how you could use Object.create():

```
const obj1 = {a: 1, b: 2};
const obj2 = Object.create(obj1);
console.log(obj2); // {}
console.log(obj2.a); // 1
```

In the example above, obj1 is the prototype object, and obj2 is the object created using Object.create(). obj2 inherits the properties of obj1, so obj2.a returns 1.

It's important to note that the properties of the prototype object are not copied to the new object. Instead, the new object has a reference to the prototype object. Any changes to the prototype object will be reflected in the new object. ### 3. Object.defineProperties() is a method in JavaScript that allows you to add new properties or modify existing properties of an object. The method takes two arguments: the object that you want to add properties to, and a descriptor object that defines the properties and their attributes.

Here's an example of how you could use Object.defineProperties():

```
const obj = {};
Object.defineProperties(obj, {
  prop1: {
    value: 1,
    writable: true
  },
  prop2: {
    value: 2,
```

```
    writable: false
  }
});
console.log(obj); // {prop1: 1, prop2: 2}
obj.prop1 = 3;
console.log(obj.prop1); // 3
obj.prop2 = 4;
console.log(obj.prop2); // 2
```

In the example above, we define two properties prop1 and prop2 using Object.defineProperties() and set their attributes such as value and writable. The value attribute specifies the value of the property, and the writable attribute specifies whether the property can be modified or not.

It's worth noting that you can also define other property attributes, such as enumerable, configurable, and get/set accessors, using Object.defineProperties(). This method allows you to fine-tune the behavior of your object properties, making it an important tool for managing objects in JavaScript. ### 4. Object.defineProperty() is a method in JavaScript that allows you to add a new property or modify an existing property of an object. The method takes three arguments: the object that you want to add a property to, the property name, and a descriptor object that defines the property's attributes.

Here's an example of how you could use Object.defineProperty():

```
const obj = {};
Object.defineProperty(obj, 'prop1', {
  value: 1,
  writable: true
});
console.log(obj); // {prop1: 1}
obj.prop1 = 3;
console.log(obj.prop1); // 3
```

In the example above, we define a property prop1 using Object.defineProperty() and set its attributes such as value and writable. The value attribute specifies the value of the property, and the writable attribute specifies whether the property can be modified or not.

It's worth noting that you can also define other property attributes, such as enumerable, configurable, and get/set accessors, using Object.defineProperty(). This method allows you to fine-tune the behavior of your object properties, making it an important tool for managing objects in JavaScript. ### 5. Object.entries() is a method in JavaScript that returns an array of arrays, where each inner array is a key-value pair from the original object. The returned array has the same iteration order as a for...in loop (the order is determined by the object's prototype chain).

Here's an example of how you could use Object.entries():

```
const obj = {a: 1, b: 2};
console.log(Object.entries(obj)); // [['a', 1], ['b', 2]]
```

In the example above, we use Object.entries() to convert the properties of the object obj into an array of arrays. Each inner array represents a key-value pair from the original object.

You can use Object.entries() to perform various operations on objects, such as converting an object to a map, looping through an object, or transforming the object into a new data structure. Additionally, Object.entries() is often used in combination with other array methods, such as map() and forEach(), to perform complex transformations on objects. ### 6. Object.freeze() is a method in JavaScript that makes an object immutable, meaning its properties cannot be changed or modified. The Object.freeze() method takes an object as its argument and returns the same object, but with its properties and values set to be read-only.

Here's an example of how you could use Object.freeze():

```
const obj = {a: 1, b: 2};
Object.freeze(obj);
obj.a = 3;
console.log(obj.a); // 1
```

In the example above, we use Object.freeze() to make the object obj immutable. Despite the attempt to change the value of obj.a to 3, the value remains 1 because the object has been frozen and its properties cannot be modified.

It's important to note that Object.freeze() only makes the object and its direct properties read-only. If an object property is an object or an array, its properties can still be modified. To make these properties also read-only, you would need to recursively freeze the object properties.

Object.freeze() is useful when you want to ensure that an object remains unchanged, either for security or data integrity reasons. For instance, you may want to freeze an object that represents the configuration of your application to prevent accidental modification of critical settings. ### 7. Object.fromEntries() is a method in JavaScript that takes an iterable object, such as an array, and returns a new object created from its key-value pairs. The iterable object must have elements that are arrays with exactly two elements, where the first element is the property key and the second element is the property value.

Here's an example of how you could use Object.fromEntries():

```
const entries = [['a', 1], ['b', 2]];
const obj = Object.fromEntries(entries);
console.log(obj); // {a: 1, b: 2}
```

In the example above, we use Object.fromEntries() to convert the array of arrays entries into an object. The first element of each inner array represents the

property key, and the second element represents the property value.

Object.fromEntries() is useful when you need to convert an array of arrays into an object, especially when you want to reverse the operation performed by Object.entries(). You can use Object.fromEntries() in conjunction with other array methods, such as map() and filter(), to perform complex transformations on arrays of arrays. ### 8. Object.getOwnPropertyDescriptor() is a method in JavaScript that returns the property descriptor for a given property of an object. The property descriptor is an object that describes the attributes of a property, such as its value, writability, enumerability, and configurability.

Here's an example of how you could use Object.getOwnPropertyDescriptor():

```
const obj = {a: 1};
const descriptor = Object.getOwnPropertyDescriptor(obj, 'a');
console.log(descriptor); // {value: 1, writable: true, enumerable: true, configurable: true
```

In the example above, we use Object.getOwnPropertyDescriptor() to retrieve the property descriptor for the property 'a' of the object obj. The returned descriptor object contains information about the property's value, writability, enumerability, and configurability.

You can use Object.getOwnPropertyDescriptor() to get information about a property, change its attributes, or create new properties with custom attributes. For instance, you may want to use Object.getOwnPropertyDescriptor() to retrieve the value of a property and then modify its writable attribute to make the property read-only.

It's important to note that Object.getOwnPropertyDescriptor() only returns the property descriptor for properties that are directly defined on the object, and not on its prototype chain. If you want to get the property descriptor for a property that is inherited from the prototype, you can use Object.getPrototypeOf() in conjunction with Object.getOwnPropertyDescriptor(). ### 9. Object.getOwnPropertyDescriptors() is a method in JavaScript that returns an object containing all the property descriptors for the properties directly defined on an object. The property descriptors are objects that describe the attributes of a property, such as its value, writability, enumerability, and configurability.

Here's an example of how you could use Object.getOwnPropertyDescriptors():

```
const obj = {a: 1, b: 2};
const descriptors = Object.getOwnPropertyDescriptors(obj);
console.log(descriptors);
// {
//   a: {value: 1, writable: true, enumerable: true, configurable: true},
//   b: {value: 2, writable: true, enumerable: true, configurable: true}
// }
```

In the example above, we use Object.getOwnPropertyDescriptors() to retrieve

all the property descriptors for the properties 'a' and 'b' of the object obj. The returned descriptors object contains all the property descriptors, each one described by an object with information about the property's value, writability, enumerability, and configurability.

You can use Object.getOwnPropertyDescriptors() to get information about all the properties of an object, change their attributes, or create new objects with custom attributes. For instance, you may want to use Object.getOwnPropertyDescriptors() to retrieve all the property descriptors for an object and then use Object.create() to create a new object with the same properties, but with modified attributes.

It's important to note that Object.getOwnPropertyDescriptors() only returns the property descriptors for properties that are directly defined on the object, and not on its prototype chain. If you want to get the property descriptors for properties that are inherited from the prototype, you can use Object.getPrototypeOf() in conjunction with Object.getOwnPropertyDescriptors(). ### 10. Object.getOwnPropertyNames() is a method in JavaScript that returns an array of all the property names (also known as keys) for the properties directly defined on an object.

Here's an example of how you could use Object.getOwnPropertyNames():

```
const obj = {a: 1, b: 2};
const keys = Object.getOwnPropertyNames(obj);
console.log(keys); // ['a', 'b']
```

In the example above, we use Object.getOwnPropertyNames() to retrieve all the property names for the properties 'a' and 'b' of the object obj. The returned keys array contains all the property names, which are strings.

You can use Object.getOwnPropertyNames() to get the names of all the properties of an object, or to iterate over all the properties of an object and perform some action on each one.

It's important to note that Object.getOwnPropertyNames() only returns the property names for properties that are directly defined on the object, and not on its prototype chain. If you want to get the property names for properties that are inherited from the prototype, you can use Object.getPrototypeOf() in conjunction with Object.getOwnPropertyNames(). ### 11. Object.getOwnPropertySymbols() is a method in JavaScript that returns an array of all the property symbols for the properties directly defined on an object. A property symbol is a unique and immutable data type in JavaScript that can be used as an object property or as a key to access properties.

Here's an example of how you could use Object.getOwnPropertySymbols():

```
const symbolJ = Symbol('j');
const symbolS = Symbol('s');
```

```
const obj = {[symbolJ]: 1, [symbolS]: 2};
const symbols = Object.getOwnPropertySymbols(obj);
console.log(symbols); // [Symbol(j), Symbol(s)]
```

In the example above, we use Object.getOwnPropertySymbols() to retrieve all the property symbols for the properties symbolJ and symbolS of the object obj. The returned symbols array contains all the property symbols, which are unique and immutable data types.

You can use Object.getOwnPropertySymbols() to get the symbols of all the properties of an object, or to iterate over all the properties of an object and perform some action on each one.

It's important to note that Object.getOwnPropertySymbols() only returns the property symbols for properties that are directly defined on the object, and not on its prototype chain. If you want to get the property symbols for properties that are inherited from the prototype, you can use Object.getPrototypeOf() in conjunction with Object.getOwnPropertySymbols(). ### 12. Object.getPrototypeOf() is a method in JavaScript that returns the prototype of an object. In other words, it returns the object that was used as the prototype when the object was created.

Here's an example of how you could use Object.getPrototypeOf():

```
const obj = {a: 1, b: 2};
const prototype = Object.getPrototypeOf(obj);
console.log(prototype); // {}
```

In the example above, we use Object.getPrototypeOf() to retrieve the prototype of the object obj. The returned prototype object is the object that was used as the prototype when obj was created.

You can use Object.getPrototypeOf() to retrieve the prototype of an object and access its properties, or to determine the structure of an object's prototype chain. The prototype chain is the chain of objects that are linked to one another through the prototype property, and it is used to provide inheritance in JavaScript.

It's important to note that Object.getPrototypeOf() only returns the immediate prototype of an object. To traverse the entire prototype chain, you will need to use a loop that repeatedly calls Object.getPrototypeOf() until you reach null ### 13. Object.hasOwnProperty() is a method in JavaScript that returns a boolean indicating whether an object has the specified property as its own property (as opposed to inheriting it).

Here's an example of how you could use Object.hasOwnProperty():

```
const obj = {a: 1, b: 2};
const hasA = obj.hasOwnProperty('a');
console.log(hasA); // true
```

```
const hasC = obj.hasOwnProperty('c');
console.log(hasC); // false
```

In the example above, we use Object.hasOwnProperty() to determine if the object obj has the properties 'a' and 'c'. The hasA variable is true because 'a' is a direct property of obj. The hasC variable is false because 'c' is not a direct property of obj.

You can use Object.hasOwnProperty() to check if an object has a particular property, or to iterate over the properties of an object and perform some action on each one.

It's important to note that Object.hasOwnProperty() only returns information about properties that are directly defined on the object, and not on its prototype chain. If you want to determine if an object has a property that is inherited from its prototype chain, you can use 'propertyName' in object instead. ### 14. Object.is() is a method in JavaScript that determines whether two values are the same value. It is similar to the equality == operator, but it has some important differences:

Object.is() considers -0 and +0 to be different, whereas == considers them to be equal. Object.is() considers NaN to be equal to itself, whereas == considers NaN to be not equal to any value, including itself. Here's an example of how you could use Object.is():

```
const a = -0;
const b = 0;
console.log(Object.is(a, b)); // false

const c = NaN;
const d = NaN;
console.log(Object.is(c, d)); // true
```

In the example above, we use Object.is() to determine if -0 and 0 are the same value, and if NaN is equal to itself. The Object.is() method returns false for the first comparison because -0 and 0 are different values, and returns true for the second comparison because NaN is equal to itself.

You can use Object.is() to compare values in your code, especially when you want to compare values for equality more strictly than the == operator. ### 15. Object.isExtensible() is a method in JavaScript that determines if an object is extensible, i.e., if new properties can be added to it. An object is extensible if it can be modified and its properties can be changed.

Here's an example of how you could use Object.isExtensible():

```
const obj = {};
console.log(Object.isExtensible(obj)); // true
```

```
Object.preventExtensions(obj);
console.log(Object.isExtensible(obj)); // false
```

In the example above, we first create an object obj and log the result of calling Object.isExtensible(obj). The result is true, which means that obj is extensible. Then we call Object.preventExtensions(obj) to make obj non-extensible. Finally, we log the result of calling Object.isExtensible(obj) again. The result is now false, which means that obj is no longer extensible.

You can use Object.isExtensible() to determine if an object can be modified, or to enforce the immutability of an object by making it non-extensible. ### 16. Object.isFrozen() is a method in JavaScript that determines if an object is frozen, i.e., if its properties cannot be changed. An object is considered frozen if all its properties are non-configurable and their values cannot be changed.

Here's an example of how you could use Object.isFrozen():

```
const obj = {};
console.log(Object.isFrozen(obj)); // false

Object.freeze(obj);
console.log(Object.isFrozen(obj)); // true
```

In the example above, we first create an object obj and log the result of calling Object.isFrozen(obj). The result is false, which means that obj is not frozen. Then we call Object.freeze(obj) to freeze obj. Finally, we log the result of calling Object.isFrozen(obj) again. The result is now true, which means that obj is now frozen and its properties cannot be changed.

You can use Object.isFrozen() to determine if an object is frozen, or to enforce the immutability of an object by freezing it. ### 17. Object.prototype.isPrototypeOf() is a method in JavaScript that determines if an object is in the prototype chain of another object. It returns true if the specified object is in the prototype chain of the object on which the method is called, and false otherwise.

Here's an example of how you could use Object.prototype.isPrototypeOf():

```
const obj1 = {};
const obj2 = Object.create(obj1);
console.log(obj1.isPrototypeOf(obj2)); // true
```

In the example above, we first create an object obj1, and then create another object obj2 using Object.create(obj1). This sets obj1 as the prototype of obj2. Then we call obj1.isPrototypeOf(obj2) to determine if obj1 is in the prototype chain of obj2. The result is true, which means that obj1 is indeed in the prototype chain of obj2.

You can use Object.prototype.isPrototypeOf() to determine the prototype chain of an object and the relationships between objects in your code. ### 18. Object.isSealed() is a method in JavaScript that determines if an object is sealed,

i.e., if its properties cannot be added or removed, and if its existing properties cannot be reconfigured. An object is considered sealed if all its properties are non-configurable and their values cannot be changed.

Here's an example of how you could use Object.isSealed():

```
const obj = {};
console.log(Object.isSealed(obj)); // false

Object.seal(obj);
console.log(Object.isSealed(obj)); // true
```

In the example above, we first create an object obj and log the result of calling Object.isSealed(obj). The result is false, which means that obj is not sealed. Then we call Object.seal(obj) to seal obj. Finally, we log the result of calling Object.isSealed(obj) again. The result is now true, which means that obj is now sealed and its properties cannot be added or removed, and their values cannot be changed.

You can use Object.isSealed() to determine if an object is sealed, or to enforce the immutability of an object by sealing it. ### 19. Object.keys() is a method in JavaScript that returns an array of the own enumerable property names of an object. It only returns the names of the properties that are directly on the object and not its prototype.

Here's an example of how you could use Object.keys():

```
const obj = { name: 'John', age: 30, city: 'New York' };
console.log(Object.keys(obj)); // ['name', 'age', 'city']
```

In the example above, we create an object obj with properties name, age, and city. Then we log the result of calling Object.keys(obj), which returns an array of the property names of obj. The result is ['name', 'age', 'city'], which means that these are the properties directly on the object obj.

You can use Object.keys() to get an array of the property names of an object, or to iterate over the properties of an object. ### 20. Object.preventExtensions() is a method in JavaScript that makes an object non-extensible, which means that new properties cannot be added to the object. Once an object is made non-extensible, it cannot be made extensible again.

Here's an example of how you could use Object.preventExtensions():

```
const obj = { name: 'John', age: 30, city: 'New York' };
console.log(Object.isExtensible(obj)); // true

Object.preventExtensions(obj);
console.log(Object.isExtensible(obj)); // false
```

In the example above, we create an object obj with properties name, age, and city. Then we log the result of calling Object.isExtensible(obj),

which returns true, which means that obj is extensible. Then we call Object.preventExtensions(obj) to make obj non-extensible. Finally, we log the result of calling Object.isExtensible(obj) again, which returns false, which means that obj is now non-extensible and new properties cannot be added to it.

You can use Object.preventExtensions() to make an object non-extensible and prevent new properties from being added to it. This can be useful in situations where you want to enforce the immutability of an object and prevent unintended modifications. ### 21. Object.prototype.propertyIsEnumerable() is a method in JavaScript that determines if an enumerable property of an object can be enumerated (i.e., included in a for...in loop or an Object.keys() call).

Here's an example of how you could use propertyIsEnumerable():

```
const obj = { name: 'John', age: 30, city: 'New York' };
console.log(obj.propertyIsEnumerable('name')); // true
console.log(obj.propertyIsEnumerable('toString')); // false
```

In the example above, we create an object obj with properties name, age, and city. Then we log the result of calling obj.propertyIsEnumerable('name'), which returns true, because the property name is enumerable. Then we log the result of calling obj.propertyIsEnumerable('toString'), which returns false, because the property toString is not enumerable.

You can use propertyIsEnumerable() to check if a property of an object is enumerable or not. This can be useful in situations where you want to filter the properties of an object that you want to enumerate or include in a loop or a list. ### 22. Object.seal() is a method in JavaScript that seals an object, making it non-extensible and preventing new properties from being added to it. However, the values of existing properties can still be changed.

Here's an example of how you could use Object.seal():

```
const obj = { name: 'John', age: 30, city: 'New York' };
Object.seal(obj);
obj.age = 35;
obj.newProp = 'hello';
console.log(obj.newProp); // undefined
console.log(Object.isSealed(obj)); // true
```

In the example above, we create an object obj with properties name, age, and city. Then we call Object.seal(obj) to seal the object, making it non-extensible. Next, we change the value of the property age to 35. This change is allowed because Object.seal() only makes the object non-extensible, not read-only. Finally, we try to add a new property newProp to the object, but it is not allowed because the object is sealed. We then use Object.isSealed() to confirm that the object is indeed sealed.

You can use Object.seal() to create objects that are not extensible, but still allow

changes to the values of existing properties. This can be useful in situations where you want to prevent the structure of an object from changing, but still allow modifications to its values. ### 23. Object.setPrototypeOf() is a method in JavaScript that allows you to set the prototype of an object to a specified object or null. The prototype is the object from which the current object inherits its properties.

Here's an example of how you could use Object.setPrototypeOf():

```
const obj1 = { name: 'John', age: 30, city: 'New York' };
const obj2 = { address: '123 Main St.', state: 'NY' };
Object.setPrototypeOf(obj1, obj2);
console.log(obj1.address); // 123 Main St.
```

In the example above, we create two objects obj1 and obj2. We then use Object.setPrototypeOf(obj1, obj2) to set the prototype of obj1 to obj2. Now obj1 inherits the properties of obj2, so we can access the property address of obj2 through obj1.

You can use Object.setPrototypeOf() to specify the prototype of an object, allowing you to reuse properties from other objects and create objects that inherit from other objects. ### 24. Object.prototype.toLocaleString() is a method in JavaScript that returns a string representation of an object, formatted according to the locale-specific conventions. The method is inherited by all objects and can be overridden by specific object types to provide locale-specific string representations.

Here's an example of how you could use toLocaleString():

```
const date = new Date();
console.log(date.toLocaleString()); // outputs the current date and time in a locale-specif
```

In the example above, we create a Date object and use its toLocaleString() method to get a string representation of the date and time, formatted according to the locale-specific conventions.

You can use toLocaleString() to format values such as dates, numbers, and currency amounts in a locale-specific way, making it easier to display the values in a way that is familiar and appropriate for the user's locale. ### 25. Object.prototype.toString() is a method in JavaScript that returns a string representation of an object. The method is inherited by all objects and can be overridden by specific object types to provide their own string representations.

Here's an example of how you could use toString():

```
const obj = { a: 1, b: 2 };
console.log(obj.toString()); // outputs "[object Object]"
```

In the example above, we create an object and use its toString() method to get a string representation of the object. By default, toString() returns the string

"[object Object]" for objects, but you can override the method to provide a custom string representation.

You can use toString() to get a string representation of an object for debugging or logging purposes. It can also be useful when you need to convert an object to a string for other purposes, such as serializing the object for storage or transmission. ### 26. Object.prototype.valueOf() is a method in JavaScript that returns the primitive value of an object. The method is inherited by all objects and can be overridden by specific object types to provide their own primitive values.

Here's an example of how you could use valueOf():

```
const obj = { a: 1, b: 2 };
console.log(obj.valueOf()); // outputs Object { a: 1, b: 2 }
```

In the example above, we create an object and use its valueOf() method to get its primitive value. By default, valueOf() returns the object itself, but you can override the method to provide a custom primitive value.

You can use valueOf() to get the primitive value of an object for comparison or manipulation purposes. For example, if you have an object that represents a date, you could use valueOf() to get the underlying timestamp value of the date. ### 27. Object.values() is a method in JavaScript that returns an array of all the enumerable property values of an object. This method returns an array in the same order as that obtained by looping over the properties of the object manually.

Here's an example of how you could use Object.values():

```
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.values(obj)); // outputs [1, 2, 3]
```

In the example above, we create an object with properties a, b, and c, and use the Object.values() method to get an array of its property values. The Object.values() method returns an array of values [1, 2, 3] in the order they were defined in the object.

You can use Object.values() to iterate over the values of an object, or to get an array of values for manipulation or analysis purposes. It is particularly useful when you want to perform operations on all values of an object, without having to manually loop over its properties. ### ***BigInt Methods*** ### 1. BigInt.asIntN() is a method in JavaScript that returns a BigInt value with a specified number of bits, where the value is represented in two's complement notation. This method is used to perform bitwise operations on BigInt values and can be used to limit the number of bits used to represent a value.

Here's an example of how you could use BigInt.asIntN():

```
const num = BigInt(0x123456789abcdef012345678n);
console.log(num.asIntN(64)); // outputs 0x123456789abcdef012345678n
```

```
console.log(num.asIntN(32)); // outputs 0x56789abcn
```

In the example above, we create a BigInt value num and use the BigInt.asIntN() method to specify the number of bits used to represent the value. In the first case, we specify 64 bits, so the output is the same as the original value. In the second case, we specify 32 bits, so the output is the lower 32 bits of the original value.

You can use BigInt.asIntN() to control the number of bits used to represent a BigInt value, especially when you need to perform bitwise operations or want to limit the number of bits used to reduce memory usage. ### 2. BigInt.asUintN() is a method in JavaScript that returns a BigInt value with a specified number of bits, where the value is represented in unsigned integer notation. This method is used to perform bitwise operations on BigInt values and can be used to limit the number of bits used to represent a value.

Here's an example of how you could use BigInt.asUintN():

```
const num = BigInt(0x123456789abcdef012345678n);
console.log(num.asUintN(64)); // outputs 0x123456789abcdef012345678n
console.log(num.asUintN(32)); // outputs 0x56789abc
```

In the example above, we create a BigInt value num and use the BigInt.asUintN() method to specify the number of bits used to represent the value. In the first case, we specify 64 bits, so the output is the same as the original value. In the second case, we specify 32 bits, so the output is the lower 32 bits of the original value, represented as an unsigned integer.

You can use BigInt.asUintN() to control the number of bits used to represent a BigInt value, especially when you need to perform bitwise operations or want to limit the number of bits used to reduce memory usage. ### ***Symbol Methods*** ### 1. Symbol.for() is a method in JavaScript that creates or retrieves a symbol with a specified key. The key is used to identify the symbol, and symbols with the same key will refer to the same value. This makes Symbol.for() useful for creating unique, shared symbols across different parts of your code.

Here's an example of how you could use Symbol.for():

```
const key = 'my_symbol';
const symbol1 = Symbol.for(key);
const symbol2 = Symbol.for(key);
console.log(symbol1 === symbol2); // true
```

In the example above, we create two symbols with the same key 'my_symbol' using the Symbol.for() method. The two symbols refer to the same value, so the comparison symbol1 === symbol2 returns true.

You can use Symbol.for() to create unique, shared symbols that can be used as property keys or for other purposes where you need a unique, shared identifier. This can be useful for avoiding naming conflicts and for implementing advanced features, such as private properties in JavaScript classes. ### 2.

Symbol.keyFor() is a method in JavaScript that returns the key of a symbol. The key is a string that was used to create the symbol with the Symbol.for() method.

Here's an example of how you could use Symbol.keyFor():

```
const key = 'my_symbol';
const symbol = Symbol.for(key);
const symbolKey = Symbol.keyFor(symbol);
console.log(symbolKey); // 'my_symbol'
```

In the example above, we create a symbol using the Symbol.for() method with the key 'my_symbol'. Then, we use the Symbol.keyFor() method to retrieve the key of the symbol. The Symbol.keyFor() method returns the key 'my_symbol', which is the same key that was used to create the symbol.

You can use Symbol.keyFor() to retrieve the key of a symbol created with Symbol.for(). This can be useful for debugging or for logging purposes, or when you need to retrieve the key of a symbol for some other reason.