1. What is the result of the following code?

```
console.log([] == ![]);
```

Answer:

- The output will be `true`.
    - `![]` is `false` because `[]` is a truthy value, and applying the logical NOT (`!`) operator results in `false`.
    - `[] == false` is true because of type coercion. JavaScript treats `[]` as falsy when compared to `false` in loose equality.

---

2. What will this code return?

```
console.log(0.1 + 0.2 == 0.3);
```

Answer:

- The output will be `false`.
    - Due to floating-point precision issues in JavaScript, `0.1 + 0.2` results in `0.30000000000000004`, which is not exactly equal to `0.3`.

---

3. What is the result of the following code?

```
let a = [];
let b = [];
console.log(a == b);
```

Answer:

- The output will be `false` because arrays in JavaScript are reference types. Even if two arrays have the same contents, they refer to different memory locations, so the comparison `a == b` checks for reference equality and returns `false`.

---

4. What is the output of this code?

```
console.log([] + []);
console.log([1] + [2]);
console.log([1, 2] + [3, 4]);
```

Answer:

- `[] + []` will return an empty string `""`.
- `[1] + [2]` will return `"12"`, as array elements are coerced to strings.
- `[1, 2] + [3, 4]` will return `"1,23,4"`, as the arrays are converted to strings and concatenated.

---

5. What will be the output of this code?

```
console.log('10' - 5);
console.log('10' + 5);
```

Answer:

- `'10' - 5` will return `5` because the `-` operator coerces the string `'10'` into a number and performs subtraction.
- `'10' + 5` will return `'105'` because the `+` operator performs string concatenation.

---

6. What will be the output of the following?

```
console.log('a' - 'b');
```

Answer:

- The output will be `NaN` because JavaScript cannot subtract two strings and returns `NaN` (Not-a-Number).

---

7. What is the output of the following?

```
console.log('0' == 0);
```

```
console.log('0' === 0);
```

Answer:

- `'0' == 0` will return `true` because JavaScript performs type coercion and converts the string `'0'` to a number before comparison.
- `'0' === 0` will return `false` because `===` checks both value and type, and `'0'` is a string while `0` is a number.

---

8. What will be the result of this code?

```
console.log([] == false);
console.log([] === false);
```

Answer:

- `[] == false` will return `true` because JavaScript coerces `[]` to `false` when using loose equality.
- `[] === false` will return `false` because `[]` is an object and `false` is a boolean, so they are not the same type.

---

9. What is the output of this code?

```
var foo = 1;
(function() {
  var foo = 2;
  console.log(foo);
})();
console.log(foo);
```

Answer:

- Inside the IIFE (Immediately Invoked Function Expression), `foo` is `2`, so it logs `2`.
- Outside the IIFE, `foo` is still `1`, so it logs `1`.

---

10. What is the result of this?

```
console.log([] == ![]);
```

Answer:

- The output will be `true`.
  - `[]` is falsy when used in a non-strict comparison, so `![]` results in `false`.
  - `[] == false` is true because of type coercion.

---

11. What will this code output?

```
console.log(typeof NaN);
```

Answer:

- The output will be `"number"`. This is a JavaScript quirk where `NaN` (Not-a-Number) is technically considered of type `number`.

---

12. What is the output of this code?

```
console.log(null == undefined);
console.log(null === undefined);
```

Answer:

- `null == undefined` will return `true` because JavaScript allows loose equality between `null` and `undefined`.
- `null === undefined` will return `false` because they are of different types (null is an object and undefined is its own type).

---

13. What will this code output?

```
const person = { name: 'Alice' };
person = { name: 'Bob' };
```

```
console.log(person);
```

Answer:

- This will throw a **TypeError**: `Assignment to constant variable`.
    - `person` is declared as a `const`, meaning its reference cannot be changed, even if its properties are mutable.

---

14. What is the output of this?

```
console.log([] instanceof Array);
console.log({} instanceof Object);
```

Answer:

- `[] instanceof Array` will return `true` because `[]` is an instance of the `Array` constructor.
- `{}` instanceof Object will return `false`. This is a tricky one. `{}` is an object literal and does not have an explicit prototype, so it doesn't pass the `instanceof Object` check.

---

15. What is the result of the following code?

```
let a = 1;
console.log(a++ + ++a);
```

Answer:

- The result will be `4`.
    - `a++` is a post-increment (returns `1`), and then `++a` increments `a` to `3` and returns the value `3`.
    - The expression becomes `1 + 3`, which is `4`.

---

16. What will be the output of the following?

```
console.log([1,2] == [1,2]);
```

Answer:

- The output will be `false` because arrays in JavaScript are reference types, and comparing two different array objects checks for reference equality, not their contents.

---

17. What will this code return?

```
const a = { x: 1, y: 2 };
const b = { x: 1, y: 2 };
console.log(a == b);
```

Answer:

- The output will be `false`. Even though `a` and `b` have the same properties and values, they are distinct objects, and JavaScript compares objects by reference, not by value.

---

18. What will this code return?

```
console.log('a' + 1 - 1);
```

Answer:

- The output will be `"a0"`.
    - `'a' + 1` results in `'a1'` (string concatenation).
    - `'a1' - 1` results in `NaN`, but JavaScript returns `'a0'` due to string coercion.

---

1. What will be the result of this?

```
console.log([] == ![]);
```

Answer:

- The result is `true`.

    - `![]` is `false` because `[]` is truthy and applying `!` makes it `false`.
    - `[] == false` is `true` due to JavaScript's type coercion: an empty array is considered falsy in loose equality.

---

2. What is the result of the following code?

```
console.log([1] == true);
console.log([0] == false);
```

Answer:

- `[1] == true` returns `true` because the array `[1]` gets converted to `"1"`, and `"1" == true` becomes `true` due to type coercion.
- `[0] == false` returns `true` because `[0]` gets converted to `"0"`, and `"0" == false` is `true`.

---

3. What is the result of this code?

```
console.log('' == 0);
console.log('' === 0);
```

Answer:

- `'' == 0` returns `true` because `""` (empty string) is falsy, and JavaScript coerces it to `0` when compared with a number.
- `'' === 0` returns `false` because strict equality (`===`) checks both type and value, and `""` is a string, while `0` is a number.

---

4. What will be the result of this code?

```
console.log([1,2] == [1,2]);
```

Answer:

- The result is `false`.

- Even though the arrays have the same values, they are different objects in memory, and JavaScript compares them by reference, not by value.

---

5. What will be the output of this?

```
let a = 0;
console.log(a++);
console.log(++a);
```

Answer:

- The output will be:
  - `0` (post-increment `a++` prints the current value of `a` before incrementing)
  - `2` (pre-increment `++a` increments `a` first, then prints the new value)

---

6. What is the output of this?

```
console.log({} + []);
```

Answer:

- The output will be `"[object Object]"`.
  - In JavaScript, an empty object `{}` is converted to the string `"[object Object]"`, and the array `[]` is an empty string `""`. The concatenation results in `"[object Object]"`.

---

7. What will be the output of the following?

```
console.log('a' + + 'b');
```

Answer:

- The result is `'aNaN'`.

    - `'b'` is a non-numeric string, so `+'b'` results in `NaN`, and `'a' + NaN` results in the string `'aNaN'`.

---

8. What will be the output of the following?

```
console.log([] + [] == false);
console.log([] + [] == "");
```

Answer:

- The first expression `[] + [] == false` is `true`.
    - `[] + []` results in an empty string, which is falsy, so `"" == false` is `true`.
- The second expression `[] + [] == ""` is also `true` because `[] + []` results in an empty string `""`.

---

9. What will be the result of this code?

```
console.log(null == undefined);
console.log(null === undefined);
```

Answer:

- `null == undefined` returns `true` because JavaScript allows loose equality between `null` and `undefined`.
- `null === undefined` returns `false` because they are of different types (`null` is an object and `undefined` is its own type).

---

10. What is the output of this?

```
console.log(typeof NaN);
```

Answer:

- The result is `"number"`.
    - Even though `NaN` stands for "Not-a-Number", it is actually of type `number` in JavaScript.

---

11. What will the following return?

```
console.log(0.1 + 0.2 == 0.3);
```

Answer:

- The result is `false`.
    - Floating-point precision issues make `0.1 + 0.2` equal to `0.30000000000000004`, not exactly `0.3`.

---

12. What will be the output of this?

```
const a = 1;
const b = a;
b = 2;
console.log(a);
```

Answer:

- The output will throw an error: `Uncaught TypeError: Assignment to constant variable`.
    - `const` variables cannot be reassigned, so the attempt to assign `2` to `b` will fail.

---

13. What will this code output?

```
console.log(typeof null);
```

Answer:

- The result is `"object"`.
    - This is a well-known JavaScript quirk: `typeof null` returns `"object"`, even though `null` is not an object.

---

14. What will be the output of the following?

```
console.log([] == []);
```

Answer:

- The result is `false`.
    - Arrays are reference types in JavaScript. Even if they contain the same elements, two different arrays are not considered equal because they refer to different memory locations.

---

15. What will be the output of this?

```
console.log([] == ![]);
```

Answer:

- The result is `true`.
    - `![]` is `false`, and `[] == false` is true because `[]` is coerced to `false` in loose equality.

---

16. What will be the result of this code?

```
console.log('0' == []);
```

Answer:

- The result is `true`.
    - `[]` is coerced to an empty string `""` in loose equality, and `'0' == ''` is `true`.

---

17. What will this code output?

```
let a = {};
console.log(a == a);
```

Answer:

- The output is `true`.
    - An object is compared by reference, and since `a` refers to the same object, it is equal to itself.

---

18. What will the following code output?

```
console.log(1 == true);
console.log(0 == false);
```

Answer:

- `1 == true` returns `true` because JavaScript coerces `true` to `1`.
- `0 == false` returns `true` because JavaScript coerces `false` to `0`.

---

19. What will the output be of this?

```
console.log('5' + 5);
console.log('5' - 5);
```

Answer:

- `'5' + 5` returns `'55'` because the `+` operator results in string concatenation.
- `'5' - 5` returns `0` because the `-` operator coerces both operands into numbers and performs subtraction.

---

20. What is the output of this?

```
console.log([1] == [1]);
```

Answer:

- The output is `false`.
    - Two separate array objects are compared by reference, and since they do not refer to the same memory location, the comparison results in `false`.

---

21. What will this code return?

```
console.log('foo' === 'foo');
```

Answer:

- The output is `true`.
    - Both strings are identical, so they are strictly equal.

---

22. What is the output of this?

```
console.log([1, 2] === [1, 2]);
```

Answer:

- The result is `false`.
    - Arrays are reference types, so even though the arrays have the same values, they are different objects in memory, and `===` checks for reference equality.

---

1. What will the following code output?

```
console.log([] == ![]);
```

Answer:

- The output will be `true`.
    - `![]` is `false` because `[]` is truthy, and `!` negates it.
    - `[] == false` is `true` because JavaScript coerces an empty array `[]` to a falsy value (`false`) when compared loosely.

---

2. What is the result of this code?

```
console.log(typeof NaN);
```

Answer:

- The result is `"number"`.
    - This is a JavaScript quirk: `NaN` (Not-a-Number) is of type `number`.

---

3. What is the output of this?

```
console.log([1,2,3] == '1,2,3');
```

Answer:

- The result is `true`.
    - When you use `==` to compare an array with a string, JavaScript implicitly converts the array to a comma-separated string, so `[1, 2, 3]` becomes `'1,2,3'`, which matches the string on the right-hand side.

---

4. What will the following code return?

```
console.log([] == ![]);
```

Answer:

- The result is `true`.
    - `![]` evaluates to `false` because `[]` is truthy, and `![]` coerces it to `false`.
    - `[] == false` is true because `[]` is coerced to `false` when compared using `==`.

---

5. What will this code return?

```
console.log(0.1 + 0.2 == 0.3);
```

Answer:

- The result is `false`.

- This is a common floating-point precision issue in JavaScript. `0.1 + 0.2` results in `0.30000000000000004`, which is not strictly equal to `0.3`.

---

6. What is the output of this?

```
console.log('10' - 1);
console.log('10' + 1);
```

Answer:

- `'10' - 1` returns `9`, because JavaScript coerces `'10'` into a number and performs the subtraction.
- `'10' + 1` returns `'101'`, because the `+` operator is for string concatenation, and JavaScript coerces `1` to a string before concatenating it with `'10'`.

---

7. What will be the output of this?

```
console.log(null == undefined);
console.log(null === undefined);
```

Answer:

- `null == undefined` is `true` because JavaScript allows loose equality between `null` and `undefined`.
- `null === undefined` is `false` because strict equality (`===`) checks both type and value, and they are different types (`null` is an object and `undefined` is its own type).

---

8. What will the following code return?

```
let a = [1,2,3];
let b = [1,2,3];
console.log(a == b);
```

Answer:

- The result is `false`.
    - Arrays in JavaScript are reference types, and they are compared by reference. Even if their contents are identical, they refer to different locations in memory.

---

9. What is the result of the following?

```
console.log([] == false);
console.log([] == true);
```

Answer:

- `[] == false` returns `true`, because `[]` is coerced to an empty string `""`, and `"" == false` is `true`.
- `[] == true` returns `false`, because `[]` is coerced to `""`, and `"" == true` is `false`.

---

10. What will be the result of this code?

```
console.log(1 + '1');
console.log(1 - '1');
```

Answer:

- `1 + '1'` returns `'11'`, because the `+` operator concatenates the number `1` with the string `'1'`, resulting in the string `'11'`.
- `1 - '1'` returns `0`, because JavaScript coerces `'1'` to a number and performs the subtraction.

---

11. What is the output of this code?

```
console.log(typeof {});
console.log(typeof []);
console.log(typeof function(){});
```

Answer:

- `typeof {}` returns `"object"`, because `{}` is an object literal in JavaScript.
- `typeof []` returns `"object"`, because arrays are technically objects in JavaScript.
- `typeof function(){}` returns `"function"`, because functions are a special type of object in JavaScript.

---

12. What will this code output?

```
console.log([1] == true);
console.log([0] == false);
```

Answer:

- `[1] == true` returns `true` because `[1]` is coerced to the string `'1'`, and `'1' == true` is `true`.
- `[0] == false` returns `true` because `[0]` is coerced to the string `'0'`, and `'0' == false` is `true`.

---

13. What will the following code return?

```
console.log(1 == [1]);
console.log(1 === [1]);
```

Answer:

- `1 == [1]` is `true`, because the array `[1]` is coerced to the string `'1'`, and `'1' == 1` is `true`.
- `1 === [1]` is `false`, because strict equality checks both value and type, and `1` is a number while `[1]` is an array.

---

14. What will this code output?

```
console.log(1 == true);
console.log(0 == false);
```

Answer:

- `1 == true` returns `true`, because JavaScript coerces `true` to `1`.
- `0 == false` returns `true`, because JavaScript coerces `false` to `0`.

---

15. What will the following return?

```
console.log([1,2,3] == '1,2,3');
```

Answer:

- The result is `true`.
    - JavaScript coerces the array `[1, 2, 3]` to the string `'1,2,3'`, which matches the string on the right side.

---

16. What is the output of this?

```
console.log('foo' + + 'bar');
```

Answer:

- The result is `'fooNaN'`.
    - `'bar'` is not a number, so `+'bar'` results in `NaN`. The concatenation with `'foo'` gives `'fooNaN'`.

---

17. What will be the result of the following?

```
console.log([1, 2] == [1, 2]);
```

Answer:

- The result is `false`.
    - Arrays are reference types, and they are compared by reference, not by value. Even though the arrays have the same values, they are different objects in memory.

---

18. What will this code output?

```
let a = {};
let b = a;
console.log(a == b);
```

Answer:

- The result is `true`.
    - Both `a` and `b` point to the same object in memory, so the comparison by reference returns `true`.

---

19. What will this code output?

```
console.log(typeof typeof 1);
```

Answer:

- The result is `"string"`.
    - `typeof 1` returns `"number"`, and `typeof "number"` returns `"string"`.

---

20. What is the result of this?

```
console.log(0.1 + 0.2 == 0.3);
```

Answer:

- The result is `false`.
    - Due to floating-point precision errors, `0.1 + 0.2` results in `0.30000000000000004`, which is not exactly equal to `0.3`.

1. What will the following code output?

```
console.log([] + []);
console.log([] + {});
```

```
console.log({} + []);
```

Answer:

- `[] + []` will output `""` (empty string) because adding two empty arrays results in an empty string.
- `[] + {}` will output `" [object Object]"` because `[]` is coerced into an empty string, and `{}` is coerced into the string `"[object Object]"`.
- `{}` + [] will throw a syntax error. This is interpreted as a block of code because the curly braces are interpreted as a code block.

---

2. What is the result of this code?

```
console.log('0' == false);
console.log('0' === false);
```

Answer:

- `'0' == false` is `true`, because JavaScript coerces both operands to numbers before comparison (`'0'` becomes `0`, and `false` becomes `0`).
- `'0' === false` is `false`, because the types are different (`'0'` is a string, and `false` is a boolean).

---

3. What will be the result of the following?

```
console.log(false == '0');
console.log(false === '0');
```

Answer:

- `false == '0'` is `true`, because `false` is coerced to `0`, and `'0'` is also coerced to `0`.
- `false === '0'` is `false`, because the types are different (`false` is a boolean and `'0'` is a string).

---

4. What will the following code output?

```
console.log([] == ![]);
```

Answer:

- The result is `true`.
  - `![]` is `false` because an empty array is truthy, and negating it makes it `false`.
  - `[] == false` is `true` because JavaScript coerces the array `[]` to a falsy value (`false`).

---

5. What does this return?

```
console.log('a' + +'b' + 'c');
```

Answer:

- The result is `"ac"`.
  - `+'b'` returns `NaN` because `'b'` is not a number.
  - The expression becomes `'a' + NaN + 'c'`, which results in `'aNaNc'`.

---

6. What is the output of this?

```
console.log([1] == true);
console.log([0] == false);
```

Answer:

- `[1] == true` is `true` because JavaScript coerces `[1]` into the string `'1'`, and `'1' == true` is `true`.
- `[0] == false` is `true` because JavaScript coerces `[0]` into the string `'0'`, and `'0' == false` is `true`.

---

7. What does this code return?

```
let a = [1, 2];
let b = a;
b.push(3);
```

```
console.log(a);
```

Answer:

- The result is `[1, 2, 3]`.
    - Since `b` is assigned to `a`, they both refer to the same array object. Modifying `b` also modifies `a` because they are both referencing the same object.

---

8. What is the result of this?

```
console.log([] == false);
console.log([] == true);
```

Answer:

- `[] == false` is `true`, because `[]` is coerced to an empty string `""`, and `"" == false` is `true`.
- `[] == true` is `false`, because `[]` is coerced to an empty string `""`, and `"" == true` is `false`.

---

9. What will this code return?

```
console.log('10' - 1);
console.log('10' + 1);
```

Answer:

- `'10' - 1` is `9`, because `-` operator coerces `'10'` to a number and subtracts `1`.
- `'10' + 1` is `'101'`, because `+` operator performs string concatenation, so `1` is coerced into a string and joined with `'10'`.

---

10. What is the output of the following?

```
console.log([] == []);
console.log([] === []);
```

Answer:

- `[] == []` is `false` because arrays are reference types, and even if they have the same contents, they point to different objects in memory.
- `[] === []` is `false` for the same reason as above.

---

11. What is the result of this?

```
console.log(typeof null);
```

Answer:

- The result is `"object"`.
    - This is a known JavaScript quirk. `typeof null` incorrectly returns `"object"`, although `null` is not an object.

---

12. What will be the result of the following?

```
console.log(0.1 + 0.2 == 0.3);
```

Answer:

- The result is `false`.
    - This is due to floating-point precision errors in JavaScript. `0.1 + 0.2` results in `0.30000000000000004`, which is not exactly equal to `0.3`.

---

13. What is the output of this?

```
console.log([] + {});
console.log({} + []);
```

Answer:

- `[] + {}` is `"[object Object]"`, because `[]` is coerced to an empty string and `{}` to the string `"[object Object]"`.

- `{}` + [] will throw a syntax error because `{}` is interpreted as a block of code, not an object.

---

14. What does this code return?

```
console.log({} == {});
```

Answer:

- The result is `false`.
    - Even though both `{}` are empty objects, they are different objects in memory, and JavaScript compares objects by reference.

---

15. What is the result of this?

```
console.log('1' == 1);
console.log('1' === 1);
```

Answer:

- `'1' == 1` is `true`, because the `==` operator performs type coercion, converting the string `'1'` to a number before comparison.
- `'1' === 1` is `false`, because the `===` operator checks both type and value, and the types are different (`string` vs. `number`).

---

16. What is the output of this?

```
console.log([1, 2, 3] == [1, 2, 3]);
```

Answer:

- The result is `false`.
    - Arrays are reference types, so even though the arrays contain the same values, they are different objects in memory.

---

17. What will be the result of this?

```
console.log([1, 2] == [1, 2]);
```

Answer:

- The result is `false` for the same reason as the previous question. Arrays are compared by reference, not by value.

---

18. What will this code return?

```
let a = {x: 1};
let b = {x: 1};
console.log(a == b);
```

Answer:

- The result is `false`.
    - Even though `a` and `b` have the same properties and values, they are different objects in memory.

---

19. What is the output of this?

```
console.log(NaN == NaN);
```

Answer:

- The result is `false`.
    - `NaN` is not equal to anything, including itself. This is defined in the ECMAScript specification.

---

20. What is the output of this?

```
console.log(0 / 0);
```

Answer:

- The result is `NaN`.
    - Dividing zero by zero in JavaScript results in `NaN` (Not-a-Number).

---

21. What will be the result of this?

```
console.log('foo' == 'foo');
console.log('foo' === 'foo');
```

Answer:

- Both are `true`.
    - The `==` and `===` operators will return `true` because the values and types are the same in both cases.

---

22. What will the following code output?

```
let a = {name: "John"};
let b = a;
let c = {name: "John"};
console.log(a == b);
console.log(a == c);
console.log(a === c);
```

Answer:

- `a == b` is `true` because `a` and `b` refer to the same object.
- `a == c` is `false` because `a` and `c` are different objects in memory, even though they have the same properties.
- `a === c` is also `false` for the same reason as above (strict comparison checks both value and type).

---

23. What is the result of this?

```
console.log(typeof (() => {}));
```

Answer:

- The result is `"function"`.
    - Arrow functions are also a special type of function, so the `typeof` operator returns `"function"`.

---

24. What will the following return?

```
console.log('2' - '1');
console.log('2' + '1');
```

Answer:

- `'2' - '1'` is `1`, because JavaScript coerces both `'2'` and `'1'` into numbers and performs subtraction.
- `'2' + '1'` is `'21'`, because the `+` operator performs string concatenation.

What will the following code output?

```
console.log([] + []);
```

1. **Answer:** `""` (empty string)

What is the result of the following?

```
console.log([] + {});
```

2. **Answer:** `"[object Object]"`

What is the output of this?

```
console.log([1] == true);
```

3. **Answer:** `true`

What will the following code return?

```
console.log(1 + '1');
```

4. **Answer:** `"11"` (String concatenation)

What is the result of this?
```
console.log(1 == true);
```

5. **Answer:** `true` (Type coercion occurs: 1 is converted to `true`)

What will this code return?
```
console.log(1 === true);
```

6. **Answer:** `false` (Strict equality, different types)

What is the output of this?
```
console.log([] == false);
```

7. **Answer:** `true` (Empty array is coerced to empty string, and `"" == false` is true)

What does this return?

```
console.log([1] == [1]);
```

8. **Answer:** `false` (Different objects in memory)

What is the result of this?
```
console.log([null] == false);
```

9. **Answer:** `false` (Array `[null]` is not the same as `false`)

What does this code return?
```
console.log(NaN == NaN);
```

10. **Answer:** `false` (NaN is not equal to itself in JavaScript)

Intermediate Level Tricky JavaScript Questions

What is the result of this?
```
console.log(0.1 + 0.2 == 0.3);
```

11. **Answer:** `false` (Floating-point precision issues)

What will this code output?
```
console.log('foo' == 'foo');
console.log('foo' === 'foo');
```

12. **Answer:** Both are `true`

What is the output of the following?

```
console.log({} == {});
```

13. **Answer:** `false` (Objects are compared by reference)

What is the result of this?

```
console.log('2' - '1');
console.log('2' + '1');
```

14. **Answer:**
    - `'2' - '1'` is `1` (numeric subtraction)
    - `'2' + '1'` is `'21'` (string concatenation)

What does this code output?

```
console.log(typeof null);
```

15. **Answer:** `"object"` (This is a well-known JavaScript quirk)

What is the result of this?

```
console.log(1 + '1' - 1);
```

16. **Answer:** `10` (String concatenation followed by numeric subtraction)

What is the result of this?

```
console.log([] + []);
console.log([] + {});
console.log({} + []);
```

17. **Answer:**
    - `[] + []` is `""`
    - `[] + {}` is `"[object Object]"`
    - `{}` + [] throws a syntax error because `{}` is treated as a block of code.

What does this code return?

```
let a = {x: 1};
let b = a;
let c = {x: 1};
console.log(a == b);
console.log(a == c);
```

18. **Answer:**
    - `a == b` is `true` (same reference)
    - `a == c` is `false` (different references)

What is the result of this?

```
console.log(1 + '1' == 11);
```

19. **Answer:** `true` (Due to type coercion, `'1'` is converted to a string)

What will this output?

```
console.log([] + []);
console.log([] == []);
```

20. **Answer:**
    - `[] + []` is `""` (empty string)
    - `[] == []` is `false` (different references)

Advanced Level Tricky JavaScript Questions

What does this code return?

```
let a = [1, 2, 3];
let b = a;
let c = [...a];
b.push(4);
console.log(a);
console.log(c);
```

21. **Answer:**
    - `a` is `[1, 2, 3, 4]` (since `b` references `a`)
    - `c` is `[1, 2, 3]` (spread operator creates a shallow copy)

What will be the result of this?

```
console.log('false' == false);
console.log('false' === false);
```

22. **Answer:**
    - `'false' == false` is `false`
    - `'false' === false` is `false`

What does the following return?

```
console.log([] == ![]);
```

23. **Answer:** `true` (Array `[]` is coerced to `false`, `![]` is also `false`)

What is the result of this?

```
let x = {a: 1};
let y = {a: 1};
console.log(x === y);
```

24. **Answer:** `false` (Different objects with the same structure)

What will the following output?

```
console.log(typeof (function() {}));
```

25. **Answer:** `"function"`

What does this return?

```
let obj = {};
let arr = [];
console.log(Array.isArray(obj));
console.log(Array.isArray(arr));
```

26. **Answer:**
    - `Array.isArray(obj)` is `false`
    - `Array.isArray(arr)` is `true`

What is the output of this code?

```
console.log(typeof NaN);
```

27. **Answer:** `"number"` (This is a JavaScript quirk)

What will this code output?

```
console.log(1 / 0);
console.log(-1 / 0);
```

28. **Answer:**

- `1 / 0` is `Infinity`
- `-1 / 0` is `-Infinity`

What is the result of the following?

```
console.log([1, 2] == [1, 2]);
```

29. **Answer:** `false` (Arrays are compared by reference)

What does this code return?

```
let x = 2;
let y = x;
y = 3;
console.log(x);
console.log(y);
```

30. **Answer:**
    - `x` is `2` (primitive value passed by value)
    - `y` is `3` (value updated for `y` only)

What will this code output?

```
console.log([] + 1 + 2);
```

31. **Answer:** `"12"` (Array `[]` is coerced to an empty string, then `1 + 2` is concatenated)

What is the result of the following?

```
console.log({} == {});
console.log({} === {});
```

32. **Answer:**
    - `{}` == `{}` is `false` (different objects)
    - `{}` === `{}` is `false` (different objects)

What is the result of this?

```
console.log('0' == false);
```

```
console.log('0' === false);
```

33. **Answer:**
    - `'0' == false` is `true` (type coercion)
    - `'0' === false` is `false` (strict equality)

What does this return?

```
console.log(Number('123') === 123);
console.log(Number('123') == 123);
```

34. **Answer:**
    - `Number('123') === 123` is `true` (both are numbers)
    - `Number('123') == 123` is `true` (both are numbers)

What is the result of this?

```
let obj = { x: 1 };
Object.freeze(obj);
obj.x = 2;
console.log(obj.x);
```

35. **Answer:** 1 (Object is frozen and cannot be modified)

What will this return?

```
console.log([] instanceof Array);
console.log([] instanceof Object);
```

36. **Answer:**
    - `[] instanceof Array` is `true`
    - `[] instanceof Object` is `true`

What is the result of the following?

```
let obj1 = {a: 1};
let obj2 = {a: 1};
console.log(obj1 == obj2);
```

37. **Answer:** `false` (Objects are compared by reference, not value)

What does this code return?

```
const foo = () => {};
const bar = function() {};
```

```
console.log(foo instanceof Object);
console.log(bar instanceof Object);
```

38. **Answer:**
    - `foo instanceof Object` is `true` (functions are objects)
    - `bar instanceof Object` is `true` (functions are objects)

What will the following return?

```
console.log(1 + true);
```

39. **Answer:** 2 (true is coerced to 1)

What will be the output of the following?

```
let a = {};
let b = a;
a.x = 2;
console.log(b.x);
```

40. **Answer:** 2 (Objects are referenced by memory address)

Advanced JavaScript Questions (Part 2)

What does the following code output?

```
console.log(NaN === NaN);
```

41. **Answer:** `false` (NaN is not equal to itself in JavaScript)

What is the result of this expression?

```
console.log({} + []);
```

42. **Answer:** `"[object Object]"` (An object converted to a string results in `[object Object]`)

What will the following return?

```
console.log('1' == 1);
console.log('1' === 1);
```

43. **Answer:**

- `'1' == 1` is `true` (coercion happens)
- `'1' === 1` is `false` (strict comparison, different types)

What does this code do?

```
console.log(Boolean(undefined));
console.log(Boolean(''));
console.log(Boolean(0));
console.log(Boolean(NaN));
console.log(Boolean({}));
console.log(Boolean([]));
```

44. **Answer:**
    - `Boolean(undefined)` is `false`
    - `Boolean('')` is `false`
    - `Boolean(0)` is `false`
    - `Boolean(NaN)` is `false`
    - `Boolean({})` is `true`
    - `Boolean([])` is `true`

What does this code return?

```
let a = 10;
let b = a++;
console.log(a);
console.log(b);
```

45. **Answer:**
    - `a` is `11` (post-increment happens after assignment)
    - `b` is `10` (value before increment)

What will this return?

```
console.log(1 == '1');
console.log(1 === '1');
```

46. **Answer:**
    - `1 == '1'` is `true` (type coercion occurs)
    - `1 === '1'` is `false` (strict equality comparison)

What does this code output?

```
let x = null;
```

```
console.log(x == undefined);
console.log(x === undefined);
```

47. **Answer:**
    - `x == undefined` is `true` (null is loosely equal to undefined)
    - `x === undefined` is `false` (strict equality comparison)

What does the following code return?

```
let arr = [1, 2, 3];
console.log(arr instanceof Array);
console.log(arr instanceof Object);
```

48. **Answer:**
    - `arr instanceof Array` is `true`
    - `arr instanceof Object` is `true` (arrays are objects in JavaScript)

What is the output of this code?

```
let x = [1, 2, 3];
let y = x;
let z = [1, 2, 3];
console.log(x === y);
console.log(x === z);
```

49. **Answer:**
    - `x === y` is `true` (same reference)
    - `x === z` is `false` (different references)

What will this code output?

```
const foo = () => {};
const bar = function() {};
console.log(foo instanceof Object);
console.log(bar instanceof Object);
```

50. **Answer:**
    - `foo instanceof Object` is `true`
    - `bar instanceof Object` is `true`

Deep Dive into Closures and Scopes

What is the output of the following?

```
function outer() {
    let a = 10;
    function inner() {
        console.log(a);
    }
    inner();
}
outer();
```

51. **Answer:** `10` (Closure captures the variable `a` from the outer function)

What will this code return?

```
function createCounter() {
    let count = 0;
    return function() {
        return count++;
    };
}
const counter = createCounter();
console.log(counter());
console.log(counter());
console.log(counter());
```

52. **Answer:**
    - First call returns `0`
    - Second call returns `1`
    - Third call returns `2`

What does the following code output?

```
var a = 1;
function test() {
    var a = 2;
    console.log(a);
}
test();
console.log(a);
```

53. **Answer:**
    - Inside `test()`, `a` is `2`
    - Outside `test()`, `a` is `1`

What is the result of this?

```
let x = 1;
function foo() {
    x = 2;
    return function bar() {
        x = 3;
        return x;
    };
}
const func = foo();
console.log(func());
console.log(x);
```

54. **Answer:**
    - `func()` returns `3`
    - `x` outside is `3` (due to the closure modifying `x`)

What is the output of the following code?

```
var x = 1;
(function() {
    var x = 2;
    (function() {
        console.log(x);
    })();
})();
```

55. **Answer:** `2` (Closures capture the variable from the closest scope)

What will the following code output?

```
let a = 10;
function closureTest() {
    console.log(a);
}
closureTest();
```

```
a = 20;
closureTest();
```

56. **Answer:**
    - First call outputs `10`
    - Second call outputs `20`

Asynchronous JavaScript (Promises, async/await)

What is the result of the following?

```
console.log(Promise.resolve(5));
```

57. **Answer:** A promise that resolves to `5`

What will this code output?

```
async function testAsync() {
    return 1;
}
testAsync().then(console.log);
```

58. **Answer:** `1` (Async functions always return a promise, which resolves to `1`)

What is the result of this code?

```
async function getData() {
    return Promise.resolve('Hello');
}
console.log(getData());
```

59. **Answer:** `Promise { <resolved>: 'Hello' }` (Returns a Promise, not the value directly)

What is the output of the following code?

```
async function foo() {
    return "foo";
}
console.log(foo());
```

60. **Answer:** `Promise { <resolved>: 'foo' }`

What is the result of this?

```
async function asyncTest() {
    await Promise.resolve("Done");
    console.log("Hello");
}
asyncTest();
```

61. **Answer:** `Done` is logged first, then `Hello`

What will this code output?

```
async function asyncExample() {
    console.log('Start');
    await new Promise(resolve => setTimeout(resolve, 1000));
    console.log('End');
}
asyncExample();
```

62. **Answer:**
    - Logs `Start`
    - After 1 second, logs `End`

What is the result of this?

```
console.log(Promise.all([Promise.resolve(1), Promise.resolve(2)]));
```

63. **Answer:** `Promise { <resolved>: [ 1, 2 ] }`

What will this code output?
javascript
CopyEdit

```
async function asyncError() {
    throw new Error("Something went wrong");
}
asyncError().catch(console.log);
```

64. **Answer:** `Error: Something went wrong`

Object Manipulation and Prototypal Inheritance

What does this code return?

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

65. **Answer:** 1 (Object is frozen, so it can't be modified)

What will be the result of this?

```
let obj1 = { a: 1 };
let obj2 = { b: 2 };
Object.assign(obj1, obj2);
console.log(obj1);
```

66. **Answer:** `{ a: 1, b: 2 }` (Shallow copy of properties from `obj2` to `obj1`)

What does this code output?

```
let person = {
    name: "John",
    greet() {
        console.log(`Hello, ${this.name}`);
    }
};
let newPerson = Object.create(person);
newPerson.name = "Doe";
newPerson.greet();
```

67. **Answer:** `Hello, Doe` (Inherited `greet()` from prototype)

What is the result of the following?

```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = function() {
    console.log(`Hello, ${this.name}`);
};
```

```
const p1 = new Person("Alice");
p1.greet();
```

68. **Answer:** `Hello, Alice` (Prototype method is called)

What will be the result of this code?

```
const obj = { a: 1 };
const newObj = Object.create(obj);
newObj.b = 2;
console.log(newObj.a);
```

69. **Answer:** `1` (Prototype chain lookup for `a`)

Array Manipulation

What is the output of this?

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

70. **Answer:** `[1, 2, 3, 4]`

JavaScript Closures, Scopes, and Context

What is the result of this code?

```
function outer() {
    let a = 10;
    return function inner() {
        console.log(a);
    };
}
const closure = outer();
closure();
```

71. **Answer:** `10` (Closure remembers the `a` variable from the outer scope)

What will the following code output?

```
function outer() {
    let a = 1;
    function inner() {
        console.log(a);
    }
    a = 2;
    return inner;
}
const closure = outer();
closure();
```

72. **Answer:** 2 (The closure captures the variable a which is updated inside the function)

What does the following code output?

```
let a = 1;
function foo() {
    let a = 2;
    console.log(a);
}
foo();
console.log(a);
```

73. **Answer:**
    - Inside `foo()`, logs 2
    - Outside `foo()`, logs 1 (outer scope variable is not affected)

What is the output of the following code?

```
var a = 1;
function test() {
    console.log(a);
    var a = 2;
}
test();
```

74. **Answer:** `undefined` (due to hoisting, a is declared but not assigned before it's logged)

What will the following code output?

```
function outer() {
    var x = 10;
    return function inner() {
        console.log(x);
    };
}
var myClosure = outer();
myClosure();
```

75. **Answer:** `10` (The closure keeps access to `x` from the outer function)

What does this code output?

```
function add() {
    let sum = 0;
    return function(num) {
        sum += num;
        return sum;
    };
}
const increment = add();
console.log(increment(1)); // 1
console.log(increment(2)); // 3
console.log(increment(3)); // 6
```

76. **Answer:** `1`, `3`, `6` (The closure remembers the `sum` value)

What will the following code output?

```
var a = 1;
function foo() {
    console.log(a);
    var a = 2;
    console.log(a);
}
foo();
```

77. **Answer:** `undefined`, `2` (Hoisting causes the first `a` to be `undefined` in `foo()`)

What is the result of this code?

```
let x = 5;
function test() {
    console.log(x);
    let x = 10;
}
test();
```

78. **Answer:** `ReferenceError: Cannot access 'x' before initialization` (Hoisting with `let` causes a Temporal Dead Zone)

Asynchronous JavaScript (Promises, Async/Await)

What will this code output?

```
async function asyncTest() {
    console.log('Start');
    await Promise.resolve('Done');
    console.log('End');
}
asyncTest();
```

79. **Answer:**
    - Logs `Start`
    - After the promise resolves, logs `End`

What is the output of this?

```
async function fetchData() {
    return 'Data fetched';
}
fetchData().then(console.log);
```

80. **Answer:** `Data fetched` (async functions return a promise, which resolves to `'Data fetched'`)

What is the output of this code?

```
setTimeout(() => {
    console.log('Hello');
```

```
}, 0);
console.log('World');
```

81. **Answer:**
    - Logs `World`
    - After a microtask, logs `Hello` (due to the asynchronous `setTimeout`)

What is the output of the following code?

```
async function test() {
    console.log('Start');
    await new Promise(resolve => resolve('Done'));
    console.log('End');
}
test();
```

82. **Answer:**
    - Logs `Start`
    - Then logs `End` after the promise resolves (`Done` is not logged since it is not explicitly output)

What does the following code output?

```
const p = new Promise((resolve, reject) => {
    resolve('Success');
    reject('Failure');
});
p.then(console.log).catch(console.log);
```

83. **Answer:** `Success` (The first `resolve` gets executed, and the `reject` is ignored)

What does this code output?

```
async function fetchData() {
    try {
        throw new Error('Something went wrong');
    } catch (error) {
        console.log(error.message);
    }
}
fetchData();
```

84. **Answer:** `Something went wrong` (The `catch` block catches and logs the error)

What is the result of this?

```
async function fetchData() {
    return 'Hello';
}
const result = await fetchData();
console.log(result);
```

85. **Answer:** `Hello` (Since the promise resolves with `'Hello'`)

Advanced Functions, Callbacks, and Bindings

What is the result of this code?

```
function greet(name, age) {
    console.log(`Hello, ${name}, you are ${age} years old`);
}
greet('Alice', 25);
```

86. **Answer:** `Hello, Alice, you are 25 years old` (Standard function invocation)

What will the following code output?

```
function greet(name = 'Guest', age = 30) {
    console.log(`Hello, ${name}, you are ${age} years old`);
}
greet();
```

87. **Answer:** `Hello, Guest, you are 30 years old` (Default values are used when arguments are not provided)

What is the result of this code?

```
function greet(name) {
    return `Hello, ${name}!`;
}
const greeting = greet.bind(null, 'Alice');
console.log(greeting());
```

88. **Answer:** `Hello, Alice!` (The `bind` method creates a new function with `Alice` bound to `name`)

What does the following code output?
javascript
CopyEdit

```
const person = {
    name: 'John',
    greet() {
        console.log(`Hello, ${this.name}`);
    }
};
const greet = person.greet;
greet();
```

89. **Answer:** `Hello, undefined` (Loses the context of `person` when calling `greet` without the `person` object)

What is the result of this code?

```
function multiply(a, b) {
    return a * b;
}
const multiplyByTwo = multiply.bind(null, 2);
console.log(multiplyByTwo(3));
```

90. **Answer:** `6` (The `bind` method creates a function that multiplies any number by 2)

Prototypal Inheritance and Object Manipulation

What will this code output?

```
function Person(name) {
    this.name = name;
}
Person.prototype.sayHello = function() {
    console.log(`Hello, ${this.name}`);
};
const p1 = new Person('Alice');
p1.sayHello();
```

91. **Answer:** `Hello, Alice` (Prototype method is used to log the message)

What does the following code output?

```
const person = {
    name: 'John',
    greet() {
        console.log(`Hello, ${this.name}`);
    }
};
const anotherPerson = Object.create(person);
anotherPerson.name = 'Alice';
anotherPerson.greet();
```

92. **Answer:** `Hello, Alice` (The object inherits the `greet` method from `person`)

What is the result of this code?

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

93. **Answer:**
    - `obj` remains `{ a: 1, b: 2 }`
    - `newObj` becomes `{ a: 1, b: 2, c: 3 }`

What will be the result of this?

```
const obj1 = { name: 'John' };
const obj2 = Object.create(obj1);
console.log(obj2.name);
```

94. **Answer:** `John` (The prototype chain allows `obj2` to inherit `name` from `obj1`)

What will this code output?

```
const person = { name: 'John' };
const anotherPerson = Object.create(person);
anotherPerson.age = 30;
console.log(anotherPerson.hasOwnProperty('name'));
```

95. **Answer:** `false` (The `name` property is inherited, so it's not `own`)

Array Methods and Functional Programming

What will this code output?

```
const arr = [1, 2, 3];
const doubled = arr.map(x => x * 2);
console.log(doubled);
```

96. **Answer:** `[2, 4, 6]`

What is the result of this code?

```
const arr = [1, 2, 3, 4];
const even = arr.filter(x => x % 2 === 0);
console.log(even);
```

97. **Answer:** `[2, 4]` (Filters out the even numbers)

What does the following code output?

```
const arr = [1, 2, 3];
const sum = arr.reduce((acc, val) => acc + val, 0);
console.log(sum);
```

98. **Answer:** `6` (Adds all elements of the array)

What is the result of this?

```
const arr = [1, 2, 3];
const flat = arr.flatMap(x => [x, x * 2]);
console.log(flat);
```

99. **Answer:** `[1, 2, 2, 4, 3, 6]` (Flattens the array after transforming each element)

Advanced JavaScript Concepts - Destructuring, Spread, Rest, and Template Literals

100. **What does the following code output?** `javascript const arr = [1, 2, 3, 4]; const [first, , third] = arr; console.log(first, third);` **Answer:** `1 3` (Destructuring skips the second element)
101. **What is the result of this?** `javascript const person = { name: 'John', age: 30 }; const { name, ...rest } = person; console.log(rest);` **Answer:** `{ age: 30 }` (Rest syntax collects remaining properties into an object)
102. **What will the following code output?** `javascript const obj = { a: 1, b: 2 }; const newObj = { ...obj, b: 3, c: 4 }; console.log(newObj);` **Answer:** `{ a: 1, b: 3, c: 4 }` (The `b` value is overwritten, and `c` is added)
103. **What is the output of this?** `javascript const arr = [1, 2, 3]; const newArr = [...arr, 4, 5]; console.log(newArr);` **Answer:** `[1, 2, 3, 4, 5]` (The spread syntax copies the elements of `arr` and adds new values)
104. **What will this code output?** `javascript const { a = 1, b = 2 } = { a: 10 }; console.log(a, b);` **Answer:** `10 2` (Destructuring uses default values when the property is undefined)
105. **What is the result of this code?** `javascript const arr = [1, 2, 3]; const [first, ...rest] = arr; console.log(first, rest);` **Answer:** `1 [2, 3]` (The rest syntax collects the remaining elements of the array)
106. **What does this code output?** `javascript const obj = { name: 'Alice', age: 25 }; const greet = `Hello, my name is ${obj.name} and I am ${obj.age} years old.`; console.log(greet);` **Answer:** `Hello, my name is Alice and I am 25 years old.` (Template literals interpolate expressions into a string)
107. **What will this code output?** `javascript const obj1 = { a: 1, b: 2 }; const obj2 = { ...obj1, a: 3 }; console.log(obj2);` **Answer:** `{ a: 3, b: 2 }` (The `a` property is overwritten by the new value)
108. **What is the output of this code?** `javascript function sum(...args) { return args.reduce((acc, val) => acc + val, 0); } console.log(sum(1, 2, 3, 4));` **Answer:** `10` (The `rest` operator collects the arguments and `reduce` calculates the sum)

JavaScript Classes and Object-Oriented Programming

109. **What is the output of this code?** `javascript class Person { constructor(name) { this.name = name; } greet() { console.log(`Hello, ${this.name}`); } } const person = new Person('Alice'); person.greet();` **Answer:** `Hello, Alice` (Object created from the `Person` class calls `greet` method)

110. **What will this code output?** ```javascript class Animal { constructor(name) { this.name = name; } speak() { console.log(${this.name} makes a sound.); } }

```
class Dog extends Animal {


   speak() {
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog('Max');
dog.speak();
```
Answer: `Max barks.` (Method `speak` is overridden in the `Dog` subclass)

111. **What will this code output?** `javascript class Person { static greet() { console.log('Hello!'); } } Person.greet();` **Answer:** `Hello!` (Static method is called on the class itself, not an instance)

112. **What will the following code output?** `javascript class Person { constructor(name) { this.name = name; } get nameUpperCase() { return this.name.toUpperCase(); } } const person = new Person('Alice'); console.log(person.nameUpperCase);` **Answer:** `ALICE` (Getter method automatically invoked when property is accessed)
113. **What does the following code output?** `javascript class Car { constructor(model) { this.model = model; } start() { console.log(`${this.model} is starting...`); } } const car = new Car('Tesla'); car.start();` **Answer:** `Tesla is starting...` (The `start` method logs the message with the model)

Advanced Topics - Performance and Optimization

114. **What is the result of this code?** `javascript let arr = [1, 2, 3]; let result = arr.filter(item => item > 2); console.log(result);` **Answer:** `[3]` (Filters out values greater than 2)

115. **What does this code output?** `javascript let arr = [1, 2, 3]; let sum = arr.reduce((acc, val) => acc + val, 0); console.log(sum);` **Answer:** `6` (The `reduce` method computes the sum of array elements)
116. **What does the following code output?** `javascript let obj = { a: 1, b: 2, c: 3 }; console.log(Object.keys(obj));` **Answer:** `['a', 'b', 'c']` (Returns an array of keys of the object)
117. **What is the result of the following?** `javascript const arr = [1, 2, 3]; const newArr = arr.map(x => x + 1); console.log(newArr);` **Answer:** `[2, 3, 4]` (The `map` method transforms each element)
118. **What will this code output?** `javascript let str = 'hello'; let newStr = str.replace('h', 'j'); console.log(newStr);` **Answer:** `jello` (The `replace` method replaces the first occurrence of `'h'` with `'j'`)

Memory Management and Garbage Collection

119. **What does this code output?** `javascript let arr = [1, 2, 3]; arr = null; console.log(arr);` **Answer:** `null` (The array is dereferenced, making it eligible for garbage collection)
120. **What is the result of this code?** `javascript let obj = { a: 1, b: 2 }; obj = null; console.log(obj);` **Answer:** `null` (The object is dereferenced and set to `null`)
121. **What does the following code output?** `javascript let obj = { a: 1, b: 2 }; obj = undefined; console.log(obj);` **Answer:** `undefined` (The object is dereferenced and set to `undefined`)
122. **What will this code output?** `javascript let x = { name: 'Alice' }; let y = x; x = null; console.log(y);` **Answer:** `{ name: 'Alice' }` (The reference `y` still points to the same object)

JavaScript Modules and Imports

123. **What is the output of this code?** ```javascript // math.js export const add = (a, b) => a + b;

```
// app.js
```

```
import { add } from './math.js';
console.log(add(2, 3));
```
Answer: `5` (The `add` function is imported and used in `app.js`)

124. **What does the following code output?** ```javascript // utils.js export const greet = name => `Hello, ${name}`;

```
// main.js
import * as utils from './utils.js';
console.log(utils.greet('Alice'));
```
Answer: `Hello, Alice` (All exports from `utils.js` are imported under the `utils` object)

125. **What is the result of this code?** `javascript // main.js import { greet } from './utils.js'; console.log(greet('Bob'));` **Answer:** `Hello, Bob` (The `greet` function is imported and used)

Asynchronous JavaScript - Promises, Async/Await, and Callbacks

126. **What is the output of this code?** `javascript const promise = new Promise((resolve, reject) => { setTimeout(() => resolve('Resolved!'), 1000); }); promise.then(console.log);` **Answer:** `Resolved!` (The promise resolves after 1 second and logs the message)
127. **What will this code output?** `javascript async function greet() { return 'Hello!'; } greet().then(console.log);` **Answer:** `Hello!` (The `async` function implicitly returns a promise, which resolves to `'Hello!'`)
128. **What is the result of this?** `javascript async function fetchData() { const data = await fetch('https://jsonplaceholder.typicode.com/posts'); const json = await data.json(); console.log(json); } fetchData();` **Answer:** Logs the data fetched from the URL as a JSON object (The function fetches data and waits for the result using `await`)

129. **What does this code output?** `javascript const promise1 = Promise.resolve(3); const promise2 = Promise.resolve(4); Promise.all([promise1, promise2]).then(console.log);` **Answer:** `[3, 4]` (The `Promise.all` method resolves when all promises are resolved)
130. **What will this code output?** `javascript const promise1 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'First')); const promise2 = new Promise((resolve, reject) => setTimeout(resolve, 200, 'Second')); Promise.race([promise1, promise2]).then(console.log);` **Answer:** `First` (The `Promise.race` method resolves as soon as the first promise resolves)
131. **What will this code output?** `javascript async function foo() { return 'foo'; } const result = foo(); console.log(result);` **Answer:** `Promise { 'foo' }` (Even though the function returns a value, it wraps it inside a promise)
132. **What is the result of this code?** `javascript function delay(ms) { return new Promise(resolve => setTimeout(resolve, ms)); } async function doSomething() { console.log('Start'); await delay(1000); console.log('End'); } doSomething();` **Answer:** `Start (after 1 second) End`

Event Loop, Stack, and Queue

133. **What is the output of this code?** `javascript console.log('Start'); setTimeout(() => console.log('Middle'), 0); console.log('End');` **Answer:** `Start End Middle` (Even though `setTimeout` has a delay of 0, the callback is placed in the event queue after the synchronous code)
134. **What will the following code output?** `javascript console.log(1); setTimeout(() => console.log(2), 0); Promise.resolve().then(() => console.log(3)); console.log(4);` **Answer:** `1 4 3 2` (Promises have higher priority than `setTimeout` in the event loop)
135. **What is the result of this code?** `javascript setTimeout(() => console.log('A'), 0); Promise.resolve().then(() => console.log('B')); setTimeout(() => console.log('C'), 0);` **Answer:** `B A C` (Promises are executed before `setTimeout` in the event loop)
136. **What does this code output?** `javascript console.log(1); setImmediate(() => console.log(2)); process.nextTick(() => console.log(3)); console.log(4);` **Answer:** `1 3 4 2` (In Node.js, `process.nextTick` runs before `setImmediate`)

Closures and Scoping

137. **What will this code output?** `javascript function outer() { let count = 0; return function inner() { count++; console.log(count); }; } const counter = outer(); counter(); counter();` **Answer:** `1 2` (The inner function has access to the `count` variable from the outer scope due to closure)
138. **What will this code output?** `javascript let count = 0; function increment() { count++; } increment(); console.log(count);` **Answer:** `1` (The global `count` variable is incremented)
139. **What is the output of this code?** `javascript function outer() { let x = 10; return function inner() { let y = 20; console.log(x + y); }; } const result = outer(); result();` **Answer:** `30` (The inner function has access to both `x` from the outer function and its own `y`)

JavaScript Memory and Performance Optimization

140. **What will the following code output?** `javascript let arr = [1, 2, 3]; arr.length = 2; console.log(arr);` **Answer:** `[1, 2]` (Modifying `length` truncates the array)
141. **What will this code output?** `javascript function fibonacci(n) { let a = 0, b = 1, sum; for (let i = 0; i < n; i++) { sum = a + b; a = b; b = sum; } return a; } console.log(fibonacci(10));` **Answer:** `55` (Calculates the 10th Fibonacci number)
142. **What is the result of this?** `javascript const arr = [1, 2, 3, 4, 5]; const sum = arr.reduce((acc, val) => acc + val, 0); console.log(sum);` **Answer:** `15` (The `reduce` method calculates the sum of the array elements)
143. **What will this code output?** `javascript function findMax(arr) { return Math.max(...arr); } console.log(findMax([1, 2, 3, 4, 5]));` **Answer:** `5` (The `Math.max` function finds the largest number in the array)

Advanced ES6+ Features

144. **What does this code output?** `javascript const nums = [1, 2, 3]; const doubled = nums.map(num => num * 2); console.log(doubled);` **Answer:** `[2, 4, 6]` (The `map` method creates a new array with each element doubled)
145. **What is the output of this code?** `javascript let person = { name: 'Alice', age: 25 }; let clone = { ...person }; clone.age = 30; console.log(person.age);` **Answer:** `25` (The original object is not mutated because the spread operator creates a shallow copy)
146. **What will this code output?** `javascript const sum = (a, b) => a + b; console.log(sum(3, 4));` **Answer:** `7` (The arrow function `sum` returns the sum of `a` and `b`)
147. **What is the result of the following?** `javascript let obj = { name: 'John', age: 30 }; Object.freeze(obj); obj.age = 31; console.log(obj.age);` **Answer:** `30` (The object is frozen, so properties cannot be modified)

Functional Programming

148. **What does this code output?** `javascript const add = (a, b) => a + b; const multiply = (a, b) => a * b; const result = [1, 2, 3].map(x => add(x, 1)).map(x => multiply(x, 2)); console.log(result);` **Answer:** `[4, 6, 8]` (First, each element is incremented by 1, and then each element is multiplied by 2)
149. **What will this code output?** `javascript const filterEvens = (arr) => arr.filter(x => x % 2 === 0); console.log(filterEvens([1, 2, 3, 4, 5]));` **Answer:** `[2, 4]` (The `filter` method returns only the even numbers)

Closures and Scopes (Deep Dive)

150. **What is a closure? Answer:** A closure is a function that retains access to its lexical scope, even when the function is executed outside of that scope. In simple terms, a closure "remembers" the environment in which it was created.

```
**Example:**

```javascript
```
```

```
function outer() {
    let count = 0;
    return function inner() {
        count++;
        console.log(count);
    };
}
const counter = outer();
counter(); // 1
counter(); // 2
```

151. **How does closure work with asynchronous code? Answer:** Closures can cause issues in asynchronous code, as the closure captures variables by reference. In cases like `setTimeout` or `setInterval`, this can lead to unexpected results if the closure references variables that change.

```
**Example:**
```javascript
function createCounter() {
    let count = 0;
    setTimeout(() => {
        console.log(count); // Output: 0, even after 1 second
    }, 1000);
}
createCounter();
```

152. **What is a lexical scope in JavaScript? Answer:** Lexical scoping refers to the visibility of variables within different parts of the code based on where they are declared. JavaScript uses lexical scoping, meaning that a function can access variables from its outer scope where it was declared.

JavaScript Design Patterns

153. **What is the Module Pattern in JavaScript? Answer:** The module pattern is used to encapsulate functionality into a unit and expose only the necessary parts via a public
```

API. It helps avoid polluting the global namespace and is commonly used for structuring code.

Example:

```javascript
const Module = (function() {
    let privateVar = 1;
    return {
        getPrivateVar: function() {
            return privateVar;
        },
        setPrivateVar: function(val) {
            privateVar = val;
        }
    };
})();

console.log(Module.getPrivateVar()); // 1
Module.setPrivateVar(2);
console.log(Module.getPrivateVar()); // 2
```

154. **What is the Singleton Pattern? Answer:** The singleton pattern ensures that a class has only one instance, and it provides a global access point to that instance.

Example:
```javascript
const Singleton = (function() {
    let instance;
    function createInstance() {
        return new Object("I am the instance");
    }
    return {
        getInstance: function() {
            if (!instance) {
```

```
                instance = createInstance();
            }
            return instance;
        }
    };
})();

const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true
```

155. **What is the Observer Pattern in JavaScript? Answer:** The observer pattern allows one object (the subject) to notify others (observers) of state changes, without the subject needing to know about the specifics of the observers.

```
**Example:**
```javascript
class Subject {
    constructor() {
        this.observers = [];
    }
    addObserver(observer) {
        this.observers.push(observer);
    }
    notify() {
        this.observers.forEach(observer => observer.update());
    }
}

class Observer {
    update() {
        console.log("State updated!");
    }
}

const subject = new Subject();
const observer = new Observer();
```
```

```
subject.addObserver(observer);
subject.notify(); // State updated!
```

Memory Management and Optimization

156. **How do closures affect memory usage in JavaScript? Answer:** Closures can increase memory usage because they retain references to variables from their outer scopes. These variables are not garbage collected as long as the closure exists. It's important to be mindful of closures when writing long-running applications.

```
**Example:**
```

```javascript
function createCounter() {
    let count = 0;
    return function increment() {
        count++;
        console.log(count);
    };
}
const counter = createCounter();
// Even though createCounter has finished execution, count is not
garbage collected due to the closure
```

157. **What is the impact of the event loop on performance? Answer:** The event loop in JavaScript handles asynchronous tasks and ensures that they are executed in the correct order. However, heavy blocking code (synchronous operations that take a long time) can block the event loop and delay the processing of asynchronous tasks. This can lead to performance issues such as UI freezes in browser-based applications.

158. **How can you optimize performance in JavaScript? Answer:** - **Minimize DOM manipulation:** Reducing the number of DOM updates improves performance. - **Debounce or throttle event listeners:** This can help reduce the frequency of function calls (e.g., in the case of `scroll` or `resize` events). - **Lazy load resources:** Load

non-critical resources only when needed. - **Use Web Workers:** For heavy computations, offload tasks to Web Workers to keep the UI responsive.

159. **How does garbage collection work in JavaScript? Answer:** JavaScript uses automatic garbage collection to manage memory. Objects are removed from memory when they are no longer referenced by any part of the code. The two main garbage collection techniques used in JavaScript are: - **Mark-and-sweep**: Marks all reachable objects and sweeps away the unreferenced ones. - **Reference counting**: Keeps track of the number of references to an object. When the reference count drops to zero, the object is collected.

JavaScript Performance Optimization

160. **What is the difference between `==` and `===` in JavaScript? Answer:** - `==` is the loose equality operator, which compares values after type coercion. - `===` is the strict equality operator, which compares both value and type without coercion.

```javascript
0 == '0';  // true (type coercion)
0 === '0'; // false (no coercion)
```

161. **What is the purpose of `requestAnimationFrame`? Answer:**
`requestAnimationFrame` is used to optimize animations by scheduling them in sync with the browser's repaint cycle. It helps create smoother animations by ensuring they are updated only when the browser is ready to repaint, rather than on every frame.

Example:
```javascript
function animate() {
    console.log('Animation frame');
    requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

162. **How does `debounce` and `throttle` differ in JavaScript? Answer:** - **Debounce:** Ensures that a function is only executed after a specified time delay has passed since the last

time it was invoked. - **Throttle:** Ensures that a function is invoked at most once in a specified period, no matter how many times the event is triggered.

```
**Example:**
- **Debounce:** Often used with search input fields to avoid triggering requests on every keystroke.
- **Throttle:** Used for events like `scroll` to limit the number of times an event handler is called.
```

Advanced JavaScript Topics

163. **What is the `Symbol` data type in JavaScript? Answer:** `Symbol` is a primitive data type introduced in ES6. It represents a unique and immutable identifier that can be used to create property keys in objects without worrying about name conflicts.

```
**Example:**
```

```javascript
const sym1 = Symbol('description');
const sym2 = Symbol('description');
console.log(sym1 === sym2); // false
```

164. **What are `BigInt` and when should you use it? Answer:** `BigInt` is a special numeric type in JavaScript that can represent integers larger than `Number.MAX_SAFE_INTEGER`. It is useful when working with arbitrarily large integers (e.g., cryptography, large databases).

```
**Example:**
```javascript
const largeNumber = BigInt(1234567890123456789012345678901234567890);
console.log(largeNumber);
```
```

165. **What is `Reflect` and how is it used? Answer:** `Reflect` is an object that provides methods for interceptable operations like property access, assignment, function calls, etc.,

similar to the functionality of `Proxy`. It helps in making code more predictable and readable by using methods for introspection and manipulation.

```
**Example:**
```javascript
const obj = { a: 1 };
Reflect.set(obj, 'b', 2);
console.log(obj); // { a: 1, b: 2 }
```
```

---

swap values

let a = 8;
let b = 9;

Destructuring Assignment (ES6)

[a, b] = [b, a];
console.log(a); // 9
console.log(b); //8

Using Arithmetic

a = a + b; // a becomes 17
b = a - b; // b becomes 8
a = a - b; // a becomes 9
console.log(a); // 9
console.log(b); // 8

Using a Temporary Variable

```
let temp = a;
a = b;
b = temp;
```

```
console.log(a); // 9
console.log(b); // 8
```

[] == [] will return false because arrays are reference types, and different arrays will not point to the same memory location

```
console.log([] == []); //false
console.log([] === []);//false same reason
```

1.null == undefined will return true because in JavaScript, null and undefined are loosely equal.

2.null === undefined will return false because === checks for both value and type, and they are of different types (null is an object, and undefined is its own type)

```
console.log(null == undefined);
console.log(null === undefined);
```

The output will be "False", because 0 is falsy in JavaScript, meaning the else block will be executed

```
var foo = 0;
if (foo) {
console.log("True");
} else{
console.log("False"); } //false
```

//result both ‘11’

```
console.log(1 + '1');
console.log('1' + 1);
```

NaN == NaN and NaN === NaN both return false. This is a unique feature in JavaScript. NaN is not equal to NaN by definition. Use Number.isNaN() to check for NaN

```
console.log( NaN == NaN );
console.log( NaN === NaN );
```

'10' - 5 will return 5 because the - operator triggers type coercion, converting the string '10' to a number and performing subtraction.
'10' + 5 will return '105' because the + operator triggers string concatenation.

```
console.log('10' - 5);
console.log('10' + 5);
```

Inside the IIFE (Immediately Invoked Function Expression), foo is 2, so it logs 2 to the console.
Outside the IIFE, foo is still 1, so it logs 1 to the console.

```
var foo = 1;
(function() {
var foo = 2;
console.log(foo);
})();
console.log(foo);
```

The output is '21' because 2 + '2' becomes '22' (string concatenation), and then '22' - 1 performs type coercion and converts '22' back to 22, resulting in 21.

```
console.log( 2 + '2' - 1 );
```

The output will be 'object'. This is a well-known quirk in JavaScript where typeof null returns 'object', even though **null is not technically an object.**

```
console.log( typeof null );
```

The output will be false. Due to floating point precision errors in JavaScript, 0.1 + 0.2 results in 0.30000000000000004, which is not exactly equal to 0.3

```
console.log(0.1 + 0.2 == 0.3);
```

```
///////
const a = {};
const b = {};
console.log(a == b);//false
```

[] + [] will return an empty string "".
[1] + [2] will return "12", as array elements are coerced to strings.
[1, 2] + [3, 4] will return "1,23,4", as the arrays are converted to strings and concatenated.

```
console.log([] + []);
console.log([1] + [2]);
console.log([1, 2] + [3, 4]);
```

The output will be NaN because JavaScript cannot subtract two strings and returns NaN (Not-a-Number).

```
console.log('a' - 'b');
```

'0' == 0 will return true because JavaScript performs type coercion and converts the string '0' to a number before comparison.
'0' === 0 will return false because === checks both value and type, and '0' is a string while 0 is a number.

```
console.log('0' == 0);
console.log('0' === 0);
```

[] == false will return true because JavaScript coerces [] to false when using loose equality.
[] === false will return false because [] is an object and false is a boolean, so they are not the same type

```
console.log([] == false);
console.log([] === false);
```

This will throw a TypeError: Assignment to constant variable.

- person is declared as a const, meaning its reference cannot be changed, even if its properties are mutable.

```
const person = { name: 'Alice' };
person = { name: 'Bob' };
console.log(person);
```

`[] instanceof Array` will return `true` because `[]` is an instance of the `Array` constructor.
`{} instanceof Object` will return `false`. This is a tricky one. `{}` is an object literal and does not have an explicit prototype, so it doesn't pass the `instanceof Object` check.

```
console.log([] instanceof Array);
console.log({} instanceof Object);
```

The result will be `4`.

- `a++` is a post-increment (returns `1`), and then `++a` increments `a` to `3` and returns the value `3`.
- The expression becomes `1 + 3`, which is `4`

```
let a = 1;
console.log(a++ + ++a);
```

- The output will be "a0".
  - 'a' + 1 results in 'a1' (string concatenation).
  - 'a1' - 1 results in NaN, but JavaScript returns 'a0' due to string coercion.

```
console.log('a' + 1 - 1);
```

The first expression [] + [] == false is true.

[] + [] results in an empty string, which is falsy, so "" == false is true.

The second expression [] + [] == "" is also true because [] + [] results in an empty string "".

```
console.log([] + [] == false);
 console.log([] + [] == "");
```

- [1] == true returns true because the array [1] gets converted to "1", and "1" == true becomes true due to type coercion.
- [0] == false returns true because [0] gets converted to "0", and "0" == false is true.

```
console.log([1] == true);
console.log([0] == false);
```

- **'' == 0 returns true because "" (empty string) is falsy, and JavaScript coerces it to 0 when compared with a number.**
- **'' === 0 returns false because strict equality (===) checks both type and value, and "" is a string, while 0 is a number.**

```
console.log('' == 0);
console.log('' === 0);
```

0 (post-increment a++ prints the current value of a before incrementing)

2 (pre-increment ++a increments a first, then prints the new value)

```
let a = 0;

console.log(a++);

console.log(++a);
```

The output will be "[object Object]".

- In JavaScript, an empty object {} is converted to the string "[object Object]", and the array [] is an empty string "". The concatenation results in "[object Object]".

```
console.log({} + []);
```

The result is 'aNaN'.

- 'b' is a non-numeric string, so +'b' results in NaN, and 'a' + NaN results in the string 'aNaN'.

```
console.log('a' + + 'b');
```

The result is true.

- [] is coerced to an empty string "" in loose equality, and '0' == '' is true.

```
console.log('0' == []);
```

Answer:

- **The output is true.**
  - **An object is compared by reference, and since a refers to the same object, it is equal to itself.**

```
let a = {};
console.log(a == a);
```

Answer:

- The output is true.
    - Both strings are identical, so they are strictly equal.

```
console.log('foo' === 'foo');
```

    -

Answer:

- The result is true.
    - When you use == to compare an array with a string, JavaScript implicitly converts the array to a comma-separated string, so [1, 2, 3] becomes '1,2,3', which matches the string on the right-hand side.

      ```
      console.log([1,2,3] == '1,2,3');
      ```

Answer:

- typeof {} returns "object", because {} is an object literal in JavaScript.
- typeof [] returns "object", because arrays are technically objects in JavaScript.

- typeof function(){} returns "function", because functions are a special type of object in JavaScript.

```
console.log(typeof {});
console.log(typeof []);
console.log(typeof function(){});
```

Answer:

- *[1] == true returns true because [1] is coerced to the string '1', and '1' == true is true.*
- *[0] == false returns true because [0] is coerced to the string '0', and '0' == false is true.*

```
console.log([1] == true);
console.log([0] == false);
```

Answer:

- The result is true.
    - Both a and b point to the same object in memory, so the comparison by reference returns true.

```
let a = {};
let b = a;
console.log(a == b);
```

Answer:

- The result is "string".
    - typeof 1 returns "number", and typeof "number" returns "string".

```
console.log(typeof typeof 1);
```

```
console.log([] + []);
console.log([] + {});
console.log({} + []);
```

- [] + [] will output "" (empty string) because adding two empty arrays results in an empty string.
- [] + {} will output " [object Object]" because [] is coerced into an empty string, and {} is coerced into the string "[object Object]".
- {} + [] will throw a syntax error. This is interpreted as a block of code because the curly braces are interpreted as a code block.

```
console.log('a' + +'b' + 'c');
```

Answer:

- The result is aNaNc.
    - +'b' returns NaN because 'b' is not a number.
    - The expression becomes 'a' + NaN + 'c', which results in 'aNaNc'.

```
console.log([] + {});
console.log({} + []);
```

Answer:

- [] + {} is "[object Object]", because [] is coerced to an empty string and {} to the string "[object Object]".

```
console.log(0 / 0)
```

- The result is NaN.
    - Dividing zero by zero in JavaScript results in NaN (Not-a-Number).

```
let a = {name: "John"};
let b = a;
let c = {name: "John"};
console.log(a == b);
console.log(a == c);
console.log(a === c);
```

Answer:

- a == b is true because a and b refer to the same object.
- a == c is false because a and c are different objects in memory, even though they have the same properties.
- a === c is also false for the same reason as above (strict comparison checks both value and type).

```
console.log(typeof (() => {}));
```

Answer:

- The result is "function".
    - Arrow functions are also a special type of function, so the typeof operator returns "function".