Short Answers

1. What is ‘use strict’? What are the major effects that it has?

`"use strict"` is a directive that makes JavaScript execute in strict mode, which helps catch errors and enforce better coding practices. It can be placed at the top of a script to enable strict mode globally or inside a function to apply it only to that function's scope. It's mainly used for writing secure code, for example it prevents the creation of undeclared global variables or duplicate parameter names.

It seems to only work if you place a `;` after it, as in `"use strict";`, otherwise it would not work as just `"use strict"`, but only in some cases, such as the following case:

```
"use strict"  // need ";" here otherwise doesn't work
(function () { console.log(this); })();
```

2. What are the different type of scopes?

The different types of scope in JavaScript include global scope, function scope, block scope, and lexical scope. Variables or functions in the global scope are accessible from anywhere in the code. Variables declared within a function are only accessible inside that function (function scope). Variables declared within a block (`if` or `for` blocks) are scoped to that block if they are declared using `let` or `const`. However, variables declared with `var` are function-scoped, not block-scoped. Lexical scope refers to the way a function can access variables from its outer (enclosing) scope, determined by where the function is defined, not where it is called.

3. What is hoisting?

Hoisting in JavaScript is the behavior where variables and function declarations are "hoisted," or moved to the top of their containing scope during the compile phase (before the runtime phase). Function declarations are not only hoisted to the top, but they are also fully initialized, so you can access and call them before their declarations in the code. Variables declared using `var` are hoisted, and if you try to access them before initialization, their value will be `undefined`. However, variables declared with `let` and `const` are hoisted but are in the temporal dead zone (TDZ), meaning that if you try to access them before their initialization, a `ReferenceError` will be thrown.

4. Explain the differences between var, let, & const.

Besides the differences mentioned above regarding hoisting, `var`, `let`, and `const` have several other differences. `var` is function-scoped, so if you declare a variable using `var` inside an `if` block within a function, you can access that variable outside the block within that function. This is not the case with `let` or `const`, as they are block-scoped. In other words, if you declare a variable inside an `if` block with `let` or `const`, you cannot access it outside that block; you cannot go from an inner block to the outer scope with `let` or `const`, but you can with var. Additionally, `var` allows you to declare the same variable name multiple times in the same scope without throwing an error, whereas `let` and `const` do not allow this. Between `let` and `const`, `let` allows you to declare a variable without initializing it (`let myVar;`), whereas `const`

requires an initial value. `let` also allows you to reassign the value of a variable (`let sum = 0; sum += 1;`), but you cannot reassign a value to a `const` variable. `var` was used before ECMAScript 6 and behaves in ways that can lead to unexpected results, so it is generally recommended to avoid using `var`, instead just use `let` and `const`.

5. What is an execution context? How does it relate to the call stack?

An execution context is the environment that contains everything needed to execute a particular block of code. There is one global execution context for the top-level code, and a new execution context is created each time a function is called. It includes the variables, scope, and the value of `this` for that code execution. The global execution context is the default environment for the code that runs at the top level of your program, outside of any function. It's the "main workspace" where the code starts running when the page or program loads. A function execution context is created whenever a function is called. It's like creating a new workspace just for that function to run, with its own set of variables and information. Once the function finishes running, that workspace is removed.

Whether in the global execution context or function execution context, there are two main phases: creation and execution. In the creation phase, JavaScript prepares everything needed to run the code, including storing references to variables and functions in memory. In the execution phase, the code actually runs, assigning values to variables and invoking functions.

The execution context is related to the call stack in JavaScript because the call stack is the mechanism that keeps track of, or stores, which execution context is currently running. Since the call stack is a "stack" data structure, it follows a "last in, first out" order. This means that the last execution context that is pushed onto the stack is the first one to be popped off. Each time a function is called, a new execution context is created for that function and is pushed onto the top of the call stack. When the function finishes executing, its execution context is popped off the stack, and the program continues with the context beneath it, which is usually the previous function or the global execution context (which is at the bottom of the stack because its the first one in).

6. What is the scope chain? How does lexical scoping work?

The scope chain is the mechanism by which JavaScript looks up the value of a variable. It starts in the current scope (the function or block where the variable is used) and moves outward through the "outer" scopes, one level at a time, until it either finds the variable or reaches the global scope. If the variable is not found in any of the outer scopes, JavaScript will throw a `ReferenceError`.

Lexical scoping refers to the way JavaScript determines the scope of a variable based on where it is physically written in the code, not where it is called. This means that a function can access variables that are defined outside of it, but not the reverse. In other words, functions have access to variables from their surrounding (or lexical) scope, but the surrounding scope does not have access to variables defined inside the function. You can also say that `let` and `const` are lexical scoped to the block in which they are declared in.

7. What is a closure? Can you provide an example use-case in words?

A closure is when a function is defined inside another function, but it is more than just that. It occurs when the inner function retains a reference to the variables and scope of its outer function even after the outer function has finished executing. This means the inner function has access to the outer function's variables

and can continue to use them, even after the outer function's execution context has been removed from the call stack. In other words, closures allow the inner function to "remember" and "access" the scope in which it was created, even after its outer function has returned.

One common use of closures is to mimic private properties or methods in a class-like structure. JavaScript doesn't have true private variables, but closure allows certain variables to be "protected" or inaccessible from outside, meaning these variables can't be directly accessed or modified by code outside the closure. However, the closure itself can still interact with these variables.

8. What is currying?

Currying is a technique in JavaScript where a function that takes multiple arguments is transformed into a sequence of functions, each of which takes a single argument. In other words, instead of calling a function with all the arguments at once, you call a series of functions, each receiving one argument at a time. Each of the nested functions in currying is a closure because it can access the arguments passed to the outer functions. Essentially, functions create other functions.

(I thought thunks (`const thunkArticleIdGet = (articleId) => async (dispatch) => {}`) were an example of currying, but I guess not.)

9. What is an IIFE? When would you use it?

An IIFE stands for Immediately-Invoked Function Expression, and it is a function that is defined and executed immediately after its creation. The key characteristic of an IIFE is that it is invoked right away, meaning the function is executed as soon as it is defined, without needing to be explicitly called later.

An IIFE is typically written using anonymous functions (functions without a name), and it's wrapped in parentheses to distinguish it from a standard function declaration. The parentheses are essential to tell JavaScript that the function is being treated as an expression. The three ways to writ them are as follows:

```
(function hello() {console.log("hello")})();
(function () {console.log("hello")})();
(() => {console.log("hello")})();
```

They are less relevant now because we can declare variables with `let` and `const`. But before `let` and `const`, they allowed you to create a private scope for variables, which keeps them from polluting the global scope, as in the example below, which isolates the variable delacred with `var`:

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i);  // will log "3" three times
  }, 1_000);
}

for (var i = 0; i < 3; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i); // will log "0", "1," and "2" as expected
```

```
    }, 1_000);
  })(i);
}
```

10. Can you name the new ES6 features?

ES6 features include: `let` and `const`; arrow functions; template literals; `Map` and `Set`; array and object destructuring; spread and rest operators; default function paramters; generators; classes; `Promises` and `async/await`; and module importing and exporting.

11. What are generators and generator functions?

A generator is an object that controls the execution of a generator function. It can be in one of two states: suspended or closed. A generator function is defined using the `function*` syntax and uses the `yield` keyword to pause its execution, yielding a value to the caller. The `.next(arg?)` method is used to resume the execution of the generator, and it returns an object containing two properties: `.value` (the value yielded by the generator) and `.done` (a boolean indicating whether the generator has finished executing). When the generator resumes, the `arg` passed to `.next()` will replace the `yield` expression. This allows for more control over the flow of execution and can be used for tasks like lazy evaluation or handling asynchronous code.

12. What is Object-Oriented Programming (OOP)?

OOP stands for Object-Oriented Programming. It is a programming paradigm based on the concept of classes, objects, and four main features: abstraction, encapsulation, inheritance, and polymorphism.

A class is a reusable template for creating objects with specified data properties and methods. An object is an instantiation, or instance, of a class. Abstraction refers to hiding the complex implementation details and showing only the essential features of an object, making it easier to interact with. Encapsulation involves bundling the data and methods that operate on that data within a class, restricting access to some of the object's internal components. Inheritance allows one class to inherit properties and methods from another, promoting code reuse. Polymorphism enables objects of different classes to be treated as objects of a common superclass, often allowing the same method or operation to behave differently depending on the object.

13. What is prototype-based OOP in JS?`

A prototype is a group of properties and methods that every object "inherits" from. All prototypes have a property `.prototype`. All classes have a property `.prototype`. All objects have a `property.__proto__` that is linked to the `class.prototype`. For example, arrays "inherit" from the `Array` prototype, which defines the array methods such as `.pop()` or `.slice()`.

It's the `.prototype` property in a class that allows an object created from that class to inherit its properties. This happens because objects created from a class have an internal `.__proto__` property, which points to the `.prototype` of the class, enabling them to access the properties and methods defined there. Simply creating a new object, such as `const myObj = {}` will also have the `.__proto__` property, and it will point to the `Object.prototype`, which is the base prototype for all JavaScript objects.

14. What is the prototype chain?

The prototype chain is the process by which JavaScript looks for an object's property. It first checks the object itself, then checks its prototype (the `.__proto__`), and continues checking the prototypes up the chain, one by one, until it either finds the property or reaches null. This is similar to the scope chain, where variables are searched in the current scope and outer scopes.

15. How do you implement inheritance in JavaScript before ES6 and with ES6?

To implement inheritance in JavaScript before ES6, it was done using constructor functions and manipulating the `prototype` property, as in the following code:

```
// create a Parent class constructor
function Person(firstname, address) {
  this.firstname = firstname;
  this.address = address;
}
// add a method to the Parent class
Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.firstname}.`);
};
// create a child class constructor
function Student(firstname, address, grade) {
  Person.call(this, firstname, address);
  this.grade = grade;
}
// inherit methods from Person by linking prototypes
Student.prototype = Object.create(Person.prototype);
// reset the constructor property of Student
Student.prototype.constructor = Student;
// add a method to the Child class
Student.prototype.study = function() {
  console.log(`${this.firstname} is studying.`);
};
// create a new instance of Student
const student1 = new Student("Alan", "123 Street", "A");
student1.sayHello();  // output: Hello, my name is Alan.
student1.study();     // output: Alan is studying.
```

ES6 introduced the class `syntax`, making inheritance easier and more readable. We define the parent class with the `class` keyword and inherit using the `extends` keyword, calling `super()` to invoke the parent class's constructor, as in the following code:

```
// create the Parent class
class Person {
  constructor(firstname, address) {
    this.firstname = firstname;
    this.address = address;
```

```
  }
  sayHello() {
    console.log(`Hello, my name is ${this.firstname}.`);
  }
}
// create the Child class that inherits from Person and call the Parent
constructor using super()
class Student extends Person {
  constructor(firstname, address, grade) {
    super(firstname, address);
    this.grade = grade;
  }
  study() {
    console.log(`${this.firstname} is studying.`);
  }
}
// cceate a new instance of Student
const student1 = new Student("Alan", "123 Street", "A");
student1.sayHello();  // output: Hello, my name is Alan.
student1.study();     // output: Alan is studying.
```

16. What does 'this' refer to in the cases that were discussed in lecture?

`this` is a special keyword used in functions to refer to a specific context. The value of `this` changes depending on where and how the function is invoked. In regular functions, `this` refers to the object from which the function was called, or the global object if the function is called outside any object. For arrow functions, `this` does not have its own context; instead, it inherits the value of `this` from the scope where the arrow function was defined.

In the lecture today, the `this` keyword was used to refer to different contexts depending on how the function was invoked. When `this` was used inside a method of an object, and the method was invoked by accessing the object, this referred to the parent object. This is because the method is directly tied to the object, as in the following code from the demo:

```
const user = {
  firstname: 'ethan',
  state: 'NJ',
  logUser: function () {
    console.log(this);
    console.log(`name: ${this.firstname}, state: ${this.state}`);
  },
  logUser2: () => {
    console.log(this);
    console.log(`name: ${this.firstname}, state: ${this.state}`);
  }
};
user.logUser();
let logFn = user.logUser;  // logFn has the reference to the function
logFn();
```

When the function was assigned to a variable and invoked separately, `this` referred to the global object (`window` in a browser), rather than the object it was originally defined in. This happens because the function loses its reference to the object when it's detached, as in the followint code from the demo:

```
console.log(this);  // global object (the window) --> {}
this.a = 2;  // setting properties on the global object
console.log(this.a);  // global object --> 2

(function () { console.log(this); })();  // global object, undefined if
strict mode
    <ref *1> Object [global] {
    global: [Circular *1],
    clearImmediate: [Function: clearImmediate],
    setImmediate: [Function: setImmediate] {
        [Symbol(nodejs.util.promisify.custom)]: [Getter]
    },
    clearInterval: [Function: clearInterval],
    clearTimeout: [Function: clearTimeout],
    setInterval: [Function: setInterval],
    setTimeout: [Function: setTimeout] {
        [Symbol(nodejs.util.promisify.custom)]: [Getter]
    },
    queueMicrotask: [Function: queueMicrotask],
    structuredClone: [Function: structuredClone],
    atob: [Getter/Setter],
    btoa: [Getter/Setter],
    performance: [Getter/Setter],
    fetch: [Function: fetch],
    navigator: [Getter],
    crypto: [Getter]
    }

(() => { console.log(this); })();  // global object --> {}
```

17. What are the differences between call, apply & bind?

The `call()` method immediately invokes the function with a given `this` value and arguments provided individually, as in the following code:

```
function greet(name) {
  console.log(`${this.greeting}, ${name}!`);
}
const person = {
  greeting: "Hello"
};
// set "this" to the "person" object
greet.call(person, "Alan");  // output: Hello, Alan!
```

The `apply()` method also immediately invokes the function with a given `this` value, but arguments are passed as an array, as in the following code:

```
function greet(name, age) {
  console.log(`${this.greeting}, ${name}! You are ${age} years old.`);
}
const person = {
  greeting: "Hello"
};
// set "this" to the "person" object
greet.apply(person, ["Alan", 18]);  // output: Hello, Alan! You are 18
years old.
```

The `bind()` method returns a new function with a permanently set `this` value and preset arguments. It does not invoke the function immediately, but instead, returns a new function that can be invoked later, as in the following code:

```
function greet(name, age) {
  console.log(`${this.greeting}, ${name}! You are ${age} years old.`);
}
const person = {
  greeting: "Hello"
};
// returns a new function where "this" is permanently set to the "person"
object
const greetPerson = greet.bind(person);
// greetPerson is a new function with "this" bound to the "person" object
greetPerson("Alan", 18);  // output: Hello, Alan! You are 18 years old.
```