**Does the interpreter resolve an object property using the scope chainor prototype chain first ?**

It uses both. When trying to resolve a property or identifier, the scope chain will be used first to locate the object. Once the object has been found, the prototype chain of that objectwill then be traversed looking for the property name.

Let’s look at an example:

var bar = {};

function foo() {

bar.a = 'Set from foo()';

return function inner() {

alert(bar.a);

}

}

foo()(); // 'Set from foo()'

Line bar.a creates the property a on the global object bar, and sets its value to 'Set from foo()'. The interpreter looks into the scope chain and as expected finds bar.a in the global context. Now, lets consider the following:

var bar = {};

function foo() {

Object.prototype.a = 'Set from prototype';

return function inner() {

alert(bar.a);

}

}

foo()(); // 'Set from prototype()'

At runtime, we invoke inner(), which tries to resolve bar.a by looking in its scope chain for the existence of bar. It finds bar in the global context, and proceeds to search bar for a property named a. However, a was never set on bar, so the interpreter traverses the object’s prototype chain and finds a was set on Object.prototype.

It is this exact behavior which explains identifier resolution; locate the object in the scope chain, then proceed up the object’s prototype chain until the property is found, or returned undefined.

<http://davidshariff.com/blog/javascript-scope-chain-and-closures/>

**Call and Apply and difference**

These two methods inherent to all functions allow you to execute any function in any desired context. This makes for incredibly powerful capabilities. The call function requires the arguments to be listed explicitly while the apply function allows you to provide the arguments as an array:

**function** user(firstName, lastName, age) {

// do something

}

user.call(window, 'John', 'Doe', 30);

user.apply(window, ['John', 'Doe', 30]);

The result of both calls is exactly the same, the user function is invoked in the context of the window and provided the same three arguments.

<http://ryanmorr.com/understanding-scope-and-context-in-javascript/>

**Event delegation**

Attach **one** event listener to the whole container, and then be able to access each item when it’s actually clicked. This is called [event delegation](https://davidwalsh.name/event-delegate), and it’s much more efficient than attaching separate event handlers.

<https://medium.freecodecamp.org/3-questions-to-watch-out-for-in-a-javascript-interview-725012834ccb>

**Debouncing**

<https://medium.freecodecamp.org/3-questions-to-watch-out-for-in-a-javascript-interview-725012834ccb>

<http://ejohn.org/blog/learning-from-twitter>

<https://css-tricks.com/debouncing-throttling-explained-examples/>

**Hoisting**

<https://medium.freecodecamp.org/function-hoisting-hoisting-interview-questions-b6f91dbc2be8>

<https://medium.freecodecamp.org/what-is-variable-hoisting-differentiating-between-var-let-and-const-in-es6-f1a70bb43d>

Functions can be conditionally declared, that is, a function statement can be nested within an IF statement, however the results are inconsistent across implementations and therefore this pattern should not be used in production code. For conditional function creation, use function expressions instead.

var hoisted = "foo" in this;

console.log(`'foo' name ${hoisted ? "is" : "is not"} hoisted. typeof foo is ${typeof foo}`);

if (false) {

function foo(){ return 1; }

}

if (true) {

function foo(){ return 1; }

}

An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation. This notation is also very useful when property names are to be dynamically determined.

**Objects**

**Object.constructor**

Returns a reference to the [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) constructor function that created the instance object. Note that the value of this property is a reference to the function itself, not a string containing the function's name. The value is only read-only for primitive values such as 1, true and "test".

**Object.prototypeOf**

prototypeObj.isPrototypeOf(object)

The object whose prototype chain will be searched.

prototypeObj prototype chain will be searched

var sum = function() { /\* put your code here \*/};  
var s = sum();  
alert(s); // 0  
alert(s(1)); // 1  
alert(s(1)(2)); //3  
alert(s(3)(4)(5)); // 12

<https://github.com/vvscode/js--interview-questions/blob/master/topics/closures.md>