### 4. Difference between “ == “ and “ === “ operators.

Both are comparison operators. The difference between both the operators is that “==” is used to compare values whereas, “ === “ is used to compare both values and types.

**Example:**

var x = 2;

var y = "2";

(x == y) // Returns true since the value of both x and y is the same

(x === y) // Returns false since the typeof x is "number" and typeof y is "string"

**The ‘ == ‘ operator compares values and not types.**  
  
While the above statement is a simple way to explain == operator, it’s not completely true  
  
The reality is that while using the ‘==’ operator, coercion takes place.  
  
The ‘==’ operator, converts both the operands to the same type and then compares them.  
  
Example:

var a = 12;

var b = "12";

### 20. Explain Closures in JavaScript.

Closures are an ability of a function to remember the variables and functions that are declared in its outer scope.

var Person = function(pName){

var name = pName;

this.getName = function(){

return name;

}

}

var person = new Person("Neelesh");

console.log(person.getName());

Let’s understand closures by example:

function randomFunc(){

var obj1 = {name:"Vivian", age:45};

return function(){

console.log(obj1.name + " is "+ "awesome"); // Has access to obj1 even when the randomFunc function is executed

}

}

var initialiseClosure = randomFunc(); // Returns a function

initialiseClosure();

Let’s understand the code above,  
  
The function randomFunc() gets executed and returns a function when we assign it to a variable:

var initialiseClosure = randomFunc();

The returned function is then executed when we invoke initialiseClosure:

initialiseClosure();

The line of code above outputs “Vivian is awesome” and this is possible because of closure.

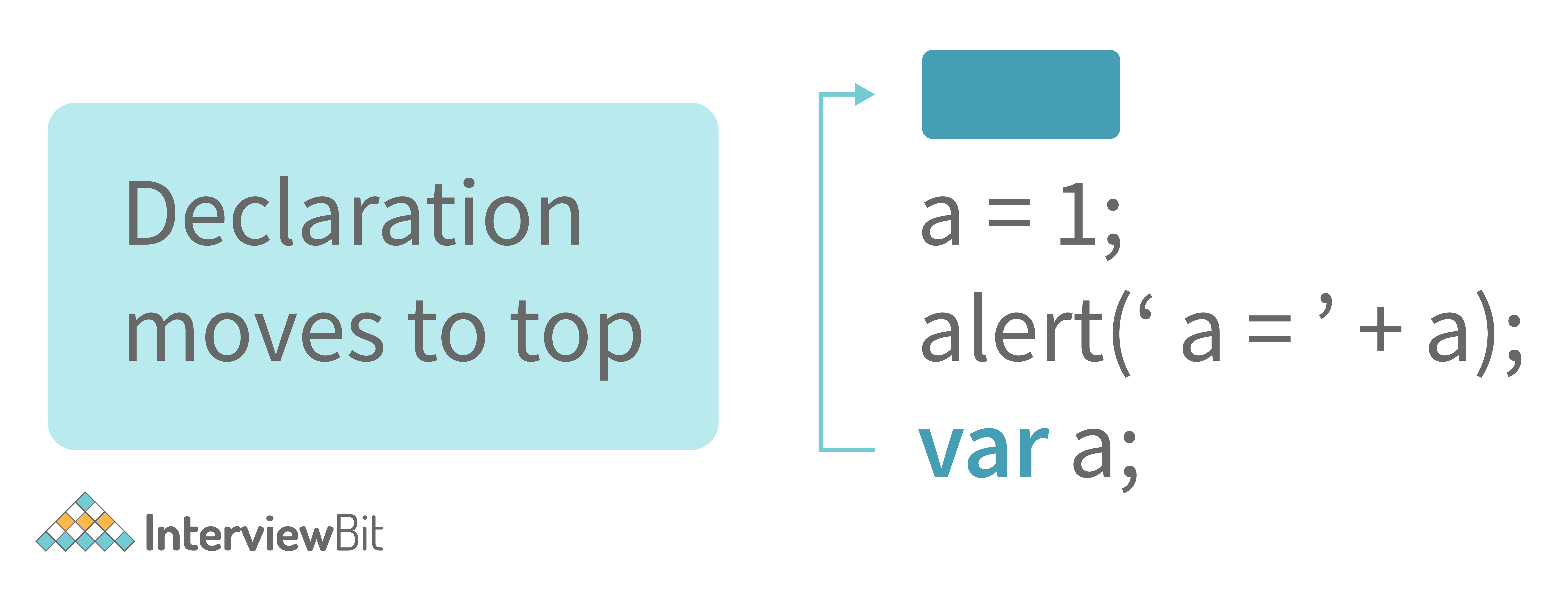
console.log(obj1.name + " is "+ "awesome");

When the function randomFunc() runs, it seems that the returning function is using the variable obj1 inside it:

Therefore randomFunc(), instead of destroying the value of obj1 after execution, **saves the value in the memory for further reference.** This is the reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed.  
  
**This ability of a function to store a variable for further reference even after it is executed is called Closure.**

### Explain Hoisting in javascript.

Hoisting is the default behaviour of javascript where all the variable and function declarations are moved on top.



This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local and global.  
  
**Example 1:**

hoistedVariable = 3;

console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initialized

var hoistedVariable;

**Example 2:**

hoistedFunction(); // Outputs " Hello world! " even when the function is declared after calling

function hoistedFunction(){

console.log(" Hello world! ");

}

**Example 3:**

// Hoisting takes place in the local scope as well

function doSomething(){

x = 33;

console.log(x);

var x;

}

doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

#### ****Note - Variable initializations are not hoisted, only variable declarations are hoisted:****

var x;

console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted

x = 23;

#### ****Note - To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code:****

"use strict";

x = 23; // Gives an error since 'x' is not declared

var x;

### 3. Why do we use the word “debugger” in javascript?

The debugger for the browser must be activated in order to debug the code. Built-in debuggers may be switched on and off, requiring the user to report faults. The remaining section of the code should stop execution before moving on to the next line while debugging

### Why do we use callbacks?

A callback function is a method that is sent as an input to another function (now let us name this other function "thisFunction"), and it is performed inside the thisFunction after the function has completed execution.

JavaScript is a scripting language that is based on events. Instead of waiting for a reply before continuing, JavaScript will continue to run while monitoring for additional events. Callbacks are a technique of ensuring that a particular code does not run until another code has completed its execution.

### 28. What is the difference between Undefined and Undeclared in JavaScript?

|  |  |
| --- | --- |
| Undefined | Undeclared |
| Undefined means a variable has been declared but a value has not yet been assigned to that variable. | Variables that are not declared or that do not exist in a program or application. |

### 29. What is the difference between Undefined and Null in JavaScript?

|  |  |
| --- | --- |
| Undefined | Null |
| Undefined means a variable has been declared but a value has not yet been assigned to that variable. | Null is an assignment value that we can assign to any variable that is meant to contain no value. |

### 12. Explain Higher Order Functions in javascript.

**Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.**  
  
Higher-order functions are a result of functions being **first-class citizens** in javascript.

Examples of higher-order functions:

function higherOrder(fn) {

fn();

}

higherOrder(function() { console.log("Hello world") });

function higherOrder2() {

return function() {

return "Do something";

}

}

var x = higherOrder2();

x() // Returns "Do something"

|  |  |
| --- | --- |
| Function declaration | Function expression |
| Declared as a separate statement within the main JavaScript code | Created inside an expression or some other construct |
| Can be called before the function is defined | Created when the execution point reaches it; can be used only after that |
| Offers better code readability and better code organization | Used when there is a need for a conditional declaration of a function |
| Example:  function abc() {      return 5;  } | Example:  var a = function abc() {      return 5;  } |

### . Differences between declaring variables using var, let and const.

|  |  |  |
| --- | --- | --- |
| var | let | const |
| There is a global scope as well as a function scope. | There is neither a global scope nor a function scope. | There is neither a global scope nor a function scope. |
| 1. There is no block scope. | There is no block scope. | There is no block scope. |
| It can be reassigned. | cIt cannot be reassigned. | It cannot be reassigned. |

Example 1: Using ‘var’ and ‘let’ variable

var variable1 = 31;

let variable2 = 89;

function catchValues()

{

  console.log(variable1);

  console.log(variable2);

// Both the variables are accessible from anywhere as their declaration is in the global scope

}

window.variable1; // Returns the value 31

window.variable2; // Returns undefined

Example 2: Using ‘const’ variable

const x = {name:"Vijay"};

x = {address: "Mumbai"}; // Throws an error

x.name = "Radha"; // No error is thrown

const y = 31;

y = 44; // Throws an error

### 60. Rest parameter and spread operator

* Rest Parameter(...)
* Rest parameter is used to declare the function with improved handling of parameters.
* Rest parameter syntax can be used to create functions to perform functions on the variable number of arguments.
* It also helps to convert any number of arguments into an array as well as helps in extracting some or all parts of the arguments.
* Spread Operator(...)
* In a function call, we use the spread operator.
* It's also to spread one or more arguments that are expected in a function call.
* The spread operator is used to take an array or an object and spread them.

### 61. Promises in JavaScript

Promises in JavaScript have four different states. They are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Pending | Fulfilled | Rejected | Settled |
| Pending is an initial state of promise. It is the initial state of promise where it is in the pending state that neither is fulfilled nor rejected. | It is the state where the promise has been fulfilled that assures that the async operation is done. | It is the state where the promise is rejected and the async operation has failed. | It is the state where the promise is rejected or fulfilled. |

Example:

function sumOfThreeElements(...elements)

{

  return new Promise((resolve,reject)=>{

    if(elements.length > 3 )

{

      reject("Just 3 elements or less are allowed");

    }

    else

{

      let sum = 0;

      let i = 0;

      while(i < elements.length)

{

        sum += elements[i];

        i++;

      }

      resolve("Sum has been calculated: "+sum);

    }

  })

}

### 37. What is the use of promises in javascript?

**Promises are used to handle asynchronous operations in javascript.**  
  
Before promises, callbacks were used to handle asynchronous operations. But due to the limited functionality of callbacks, using multiple callbacks to handle asynchronous code can lead to unmanageable code.  
  
Promise object has four states -

* Pending - Initial state of promise. This state represents that the promise has neither been fulfilled nor been rejected, it is in the pending state.
* Fulfilled - This state represents that the promise has been fulfilled, meaning the async operation is completed.
* Rejected - This state represents that the promise has been rejected for some reason, meaning the async operation has failed.
* Settled - This state represents that the promise has been either rejected or fulfilled.

A promise is created using the **Promise** constructor which takes in a callback function with two parameters, **resolve** and **reject** respectively.

### Difference between Async/Await and Generators usage to achieve the same functionality.

* Generator functions are run by their generator yield by yield which means one output at a time, whereas Async-await functions are executed sequentially one after another.
* Async/await provides a certain use case for Generators easier to execute.
* The output result of the Generator function is always value: X, done: Boolean, but the return value of the Async function is always an assurance or throws an error.

### 74. What is Lexical Scoping?

Lexical Scoping in JavaScript can be performed when the internal state of the JavaScript function object consists of the function’s code as well as references concerning the current scope chain.

### 71. Difference between Async/Await and Generators

* Async/Await
* Async-await functions are executed sequentially one after another in an easier way.
* Async/Await function might throw an error when the value is returned.
* Generators
* Generator functions are executed with one output at a time by the generator’s yield by yield.
* The ‘value: X, done: Boolean’ is the output result of the Generator function.

### 88. How do you remove duplicates from a JavaScript array?

There are two ways in which we can remove duplicates from a JavaScript array:

#### By Using the Filter Method

To call the [filter() method](https://www.simplilearn.com/tutorials/javascript-tutorial/arry-filter-in-javascript), three arguments are required. These are namely array, current element, and index of the current element.

#### By Using the For Loop

An empty array is used for storing all the repeating elements.

3. different types of variables keywords and diff b/w them --- answered.

These are the different types of data that JavaScript supports:

* Boolean - For true and false values
* Null - For empty or unknown values
* Undefined - For variables that are only declared and not defined or initialized

### What is the difference between Undefined and Null in JavaScript?

|  |  |
| --- | --- |
| Undefined | Null |
| Undefined means a variable has been declared but a value has not yet been assigned to that variable. | Null is an assignment value that we can assign to any variable that is meant to contain no value. |

* Number - For integer and floating-point numbers
* String - For characters and alphanumeric values
* Object - For collections or complex values
* Symbols - For unique identifiers for objects

### . What is currying in JavaScript?

**Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.**

Example of a curried function:

function add (a) {

return function(b){

return a + b;

}

}

add(3)(4)

For Example, if we have a function **f(a,b)**, then the function after currying, will be transformed to **f(a)(b).**  
  
By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked.  
  
Let’s see currying in action:

function multiply(a,b){

return a\*b;

}

function currying(fn){

return function(a){

return function(b){

return fn(a,b);

}

}

}

var curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12

As one can see in the code above, we have transformed the function **multiply(a,b)** to a function **curriedMultiply** , which takes in one parameter at a time.

### . Explain Implicit Type Coercion in javascript.

Implicit type coercion in javascript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

* **String coercion**

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

var x = 3;

var y = "3";

x + y // Returns "33"

Example 2:

var x = 24;

var y = "Hello";

x + y // Returns "24Hello";

#### Note - ‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings, outputs the concatenated string:

var name = "Vivek";

var surname = " Bisht";

name + surname // Returns "Vivek Bisht"

Let’s understand both the examples where we have added a number to a string,

When JavaScript sees that the operands of the expression x + y are of different types ( one being a number type and the other being a string type ), it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.

#### Note - Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, a string is converted to a number and then subtraction takes place.

var x = 3;

Var y = "3";

x - y //Returns 0 since the variable y (string type) is converted to a number type

* **Boolean Coercion**

Boolean coercion takes place when using logical operators, ternary operators, if statements, and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.  
  
Truthy values are those which will be converted (coerced) to **true**. Falsy values are those which will be converted to **false**.  
  
All values except **false, 0, 0n, -0, “”, null, undefined, and NaN** are truthy values.

**If statements:**

Example:

var x = 0;

var y = 23;

if(x) { console.log(x) } // The code inside this block will not run since the value of x is 0(Falsy)

if(y) { console.log(y) } // The code inside this block will run since the value of y is 23 (Truthy)

* **Logical operators:**

Logical operators in javascript, unlike operators in other programming languages, **do not return true or false. They always return one of the operands.**  
  
**OR ( | | ) operator** - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.  
  
**AND ( && ) operator** - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.  
  
Example:

var x = 220;

var y = "Hello";

var z = undefined;

x | | y // Returns 220 since the first value is truthy

x | | z // Returns 220 since the first value is truthy

x && y // Returns "Hello" since both the values are truthy

y && z // Returns undefined since the second value is falsy

if( x && y ){

console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)

}

if( x || z ){

console.log("Code runs"); // This block runs because x || y returns 220(Truthy)

}

* **Equality Coercion**

Equality coercion takes place when using ‘ == ‘ operator. As we have stated before  
  
**The ‘ == ‘ operator compares values and not types.**  
  
While the above statement is a simple way to explain == operator, it’s not completely true  
  
The reality is that while using the ‘==’ operator, coercion takes place.  
  
The ‘==’ operator, converts both the operands to the same type and then compares them.  
  
Example:

var a = 12;

var b = "12";

a == b // Returns true because both 'a' and 'b' are converted to the same type and then compared. Hence the operands are equal.

Coercion does not take place when using the ‘===’ operator. Both operands are not converted to the same type in the case of ‘===’ operator.

Example:

var a = 226;

var b = "226";

a === b // Returns false because coercion does not take place and the operands are of different types. Hence they are not equal.

### 15. Explain call(), apply() and, bind() methods.

**1. call():**

* It’s a predefined method in javascript.
* This method invokes a method (function) by specifying the owner object.
* Example 1:

function sayHello(){

return "Hello " + this.name;

}

var obj = {name: "Sandy"};

sayHello.call(obj);

// Returns "Hello Sandy"

* call() method allows an object to use the method (function) of another object.
* Example 2:

var person = {

age: 23,

getAge: function(){

return this.age;

}

}

var person2 = {age: 54};

person.getAge.call(person2);

// Returns 54

* call() accepts arguments:

function saySomething(message){

return this.name + " is " + message;

}

var person4 = {name: "John"};

saySomething.call(person4, "awesome");

// Returns "John is awesome"

**apply()**  
  
The apply method is similar to the call() method. The only difference is that,  
  
**call() method takes arguments separately whereas, apply() method takes arguments as an array.**

function saySomething(message){

return this.name + " is " + message;

}

var person4 = {name: "John"};

saySomething.apply(person4, ["awesome"]);

**2. bind():**

* This method returns a new function, where the value of **“this”** keyword will be bound to the owner object, which is provided as a parameter.
* Example with arguments:

var bikeDetails = {

displayDetails: function(registrationNumber,brandName){

return this.name+ " , "+ "bike details: "+ registrationNumber + " , " + brandName;

}

}

var person1 = {name: "Vivek"};

var detailsOfPerson1 = bikeDetails.displayDetails.bind(person1, "TS0122", "Bullet");

// Binds the displayDetails function to the person1 object

detailsOfPerson1();

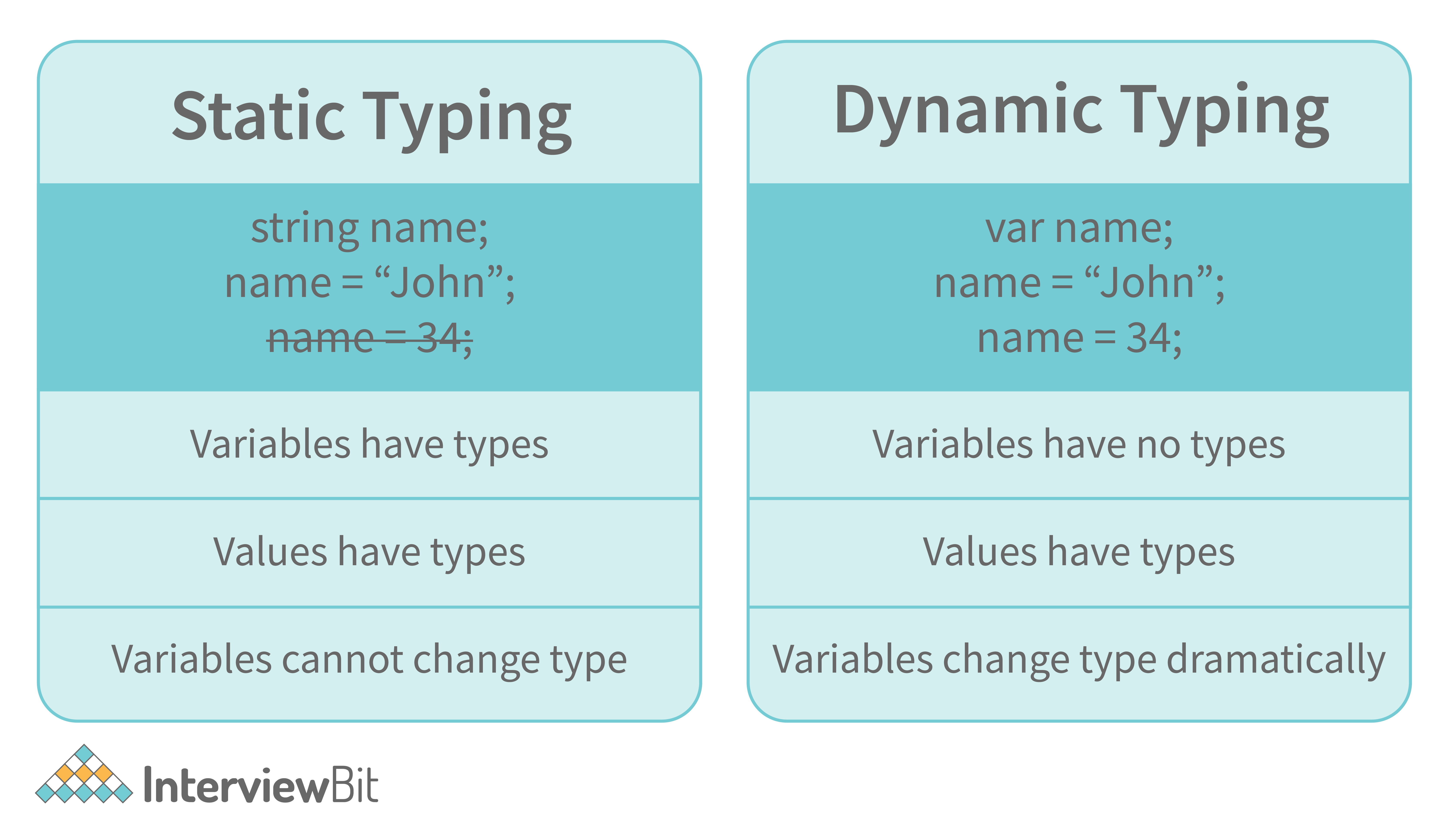
// Returns Vivek, bike details: TS0452, Thunderbird

### 16. What is the difference between exec () and test () methods in javascript?

* **test ()** and **exec ()** are RegExp expression methods used in javascript.
* We'll use **exec ()** to search a string for a specific pattern, and if it finds it, it'll return the pattern directly; else, it'll return an 'empty' result.
* We will use a **test ()** to find a string for a specific pattern. It will return the Boolean value 'true' on finding the given text otherwise, it will return 'false'.

### 7. Is javascript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during **run-time** in contrast to a statically typed language, where the type of a variable is checked during **compile-time.**



Since javascript is a loosely(dynamically) typed language, variables in JS are not associated with any type. A variable can hold the value of any data type.

For example, a variable that is assigned a number type can be converted to a string type:

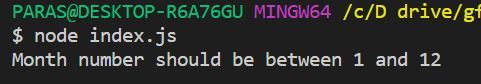
var a = 23;

var a = "Hello World!";

### 8. What is NaN property in JavaScript?

|  |
| --- |
| var monthNumber = 14;    if (monthNumber < 1 || monthNumber > 12) {        // Assigning monthNumber NaN as      // month number is not valid      monthNumber = Number.NaN;        console.log("Month number should be"                  + " between 1 and 12");  }  else {      console.log(monthNumber);  } |

**Output:**



NaN property represents the **“Not-a-Number”** value. It indicates a value that is not a legal number.

**typeof** of NaN will return a **Number**.

To check if a value is NaN, we use the **isNaN()** function,

#### Note- isNaN() function converts the given value to a Number type, and then equates to NaN.

isNaN("Hello") // Returns true

isNaN(345) // Returns false

isNaN('1') // Returns false, since '1' is converted to Number type which results in 0 ( a number)

isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)

isNaN(false) // Returns false

isNaN(undefined) // Returns true

### Pure functions

Pure functions always returns the **same result** if the same arguments are passed in. It does not depend on any state, or data, change during a program's execution. It must only depend on its **input arguments** . They do not have any side effects like network or database calls and do not modify tfunction getSquare(x) {

return x \* x;

}he arguments which are passed to them.

#### example

### Impure functions

Any function that changes the **internal state** of one of its arguments or the value of some external variable is an **impure function** . They may have any side effects like network or database calls and it may modify the arguments which are passed to them.

#### example

function getSquare(items) {

var len = items.length;

for (var i = 0; i < len; i++) {

items[i] = items[i] \* items[i];

}

return items;

eck the definition again:

### 13. Explain “this” keyword.

**The “this” keyword refers to the object that the function is a property of.**  
  
**The value of the “this” keyword will always depend on the object that is invoking the function.\**

Confused? Let’s understand the above statements by examples:

function doSomething() {

console.log(this);

}

doSomething();

What do you think the output of the above code will be?

Note - Observe the line where we are invoking the function.

Check the definition again:

#### ****The “this” keyword refers to the object that the function is a property of.****

In the above code, the function is a property of which object?

Since the function is invoked in the global context, **the function is a property of the global object.**

Therefore, the output of the above code will be **the global object.** Since we ran the above code inside the browser, the global object is **the window object.**

Example 2:

var obj = {

name: "vivek",

getName: function(){

console.log(this.name);

}

}

obj.getName();

In the above code, at the time of invocation, the getName function is a property of the object **obj** , therefore, **this** keyword will refer to the object **obj**, and hence the output will be “vivek”.

Example 3:

var obj = {

name: "vivek",

getName: function(){

console.log(this.name);

}

}

var getName = obj.getName;

var obj2 = {name:"akshay", getName };

obj2.getName();

Can you guess the output here?

The output will be “akshay”.

Although the getName function is declared inside the object **obj**, at the time of invocation, getName() is a property of **obj2**, therefore the “this” keyword will refer to **obj2**.

The silly way to understand the “**this”** keyword is, whenever the function is invoked, check the object before the **dot**. The value of **this** . keyword will always be the object before the **dot**.

If there is no object before the dot-like in example1, the value of this keyword will be the global object.

Example 4:

var obj1 = {

address : "Mumbai,India",

getAddress: function(){

console.log(this.address);

}

}

var getAddress = obj1.getAddress;

var obj2 = {name:"akshay"};

obj2.getAddress();

Can you guess the output?

**The output will be an error.**

Although in the code above, this keyword refers to the object **obj2**, obj2 does not have the property “address”‘, hence the getAddress function throws an error.

### 14. What do you mean by Self Invoking Functions?

Without being requested, a self-invoking expression is automatically invoked (initiated). If a function expression is followed by (), it will execute automatically. A function declaration cannot be invoked by itself.

Normally, we declare a function and call it, however, anonymous functions may be used to run a function automatically when it is described and will not be called again. And there is no name for these kinds of functions.

### . What is an Immediately Invoked Function in JavaScript?

**An Immediately Invoked Function ( known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.**

Syntax of IIFE :

(function(){

// Do something;

})();

To understand IIFE, we need to understand the two sets of parentheses that are added while creating an IIFE :  
  
The first set of parenthesis:

(function (){

//Do something;

})

While executing javascript code, whenever the compiler sees the word “function”, it assumes that we are declaring a function in the code. Therefore, if we do not use the first set of parentheses, the compiler throws an error because it thinks we are declaring a function, and by the syntax of declaring a function, a function should always have a name.

function() {

//Do something;

}

// Compiler gives an error since the syntax of declaring a function is wrong in the code above.

To remove this error, we add the first set of parenthesis that tells the compiler that the function is not a function declaration, instead, it’s a function expression.  
  
The second set of parenthesis:

(function (){

//Do something;

})();

From the definition of an IIFE, we know that our code should run as soon as it is defined. A function runs only when it is invoked. If we do not invoke the function, the function declaration is returned:

(function (){

// Do something;

})

// Returns the function declaration

**Therefore to invoke the function, we use the second set of parenthesis.**