**Question 1 :**

**Explore and explain the various methods in console function**

**Explain them Ex. console.log() console.warn() etc...**

Ans:- The console object provides access to the browser's debugging consol.The console object can be accessed

from any global object. Window on browsing scopes and WorkerGlobalScope as specific variants in workers via the

property console.It's exposed as Window.console, and can be referenced as simply console.

for eg. console.log("error found").

**Methods**

**console.assert()**

Log a message and stack trace to console if the first argument is false.

**console.clear()**

Clear the console.

**console.count()**

Log the number of times this line has been called with the given label.

**console.countReset()**

Resets the value of the counter with the given label.

**console.debug()**

Outputs a message to the console with the log level "debug".

**console.dir()**

Displays an interactive listing of the properties of a specified JavaScript object. This listing lets you use disclosure triangles to examine the contents of child objects.

**console.dirxml()**

Displays an XML/HTML Element representation of the specified object if possible or the JavaScript Object view if it is not possible.

**console.error()**

Outputs an error message. You may use string substitution and additional arguments with this method.

**console.exception()**

An alias for error().

**console.group()**

Creates a new inline group, indenting all following output by another level. To move back out a level, call groupEnd().

**console.groupCollapsed()**

Creates a new inline group, indenting all following output by another level. However, unlike group() this starts with the inline group collapsed requiring the use of a disclosure button to expand it. To move back out a level, call groupEnd().

**console.groupEnd()**

Exits the current inline group.

**console.info()**

Informative logging of information. You may use string substitution and additional arguments with this method.

**console.log()**

For general output of logging information. You may use string substitution and additional arguments with this method.

**console.profile()**

Starts the browser's built-in profiler (for example, the Firefox performance tool). You can specify an optional name for the profile.

**console.profileEnd()**

Stops the profiler. You can see the resulting profile in the browser's performance tool (for example, the Firefox performance tool).

**console.table()**

Displays tabular data as a table.

**console.time()**

Starts a timer with a name specified as an input parameter. Up to 10,000 simultaneous timers can run on a given page.

**console.timeEnd()**

Stops the specified timer and logs the elapsed time in seconds since it started.

**console.timeLog()**

Logs the value of the specified timer to the console.

**console.timeStamp()**

Adds a marker to the browser's Timeline or Waterfall tool.

**console.trace()**

Outputs a stack trace.

**console.warn()**

Outputs a warning message. You may use string substitution and additional arguments with this method.

**Question 2 : Write the difference between var, let and const with code examples.**

Ans:-

**var** vs **let** vs **const**

**var:** function scoped

undefined when accessing a variable before it's declared

**let:** block scoped

ReferenceError when accessing a variable before it's declared

**const:** block scoped

ReferenceError when accessing a variable before it's declare can't be reassigned

First, let's compare var and let. The main difference between var and let is that **instead of being function scoped**, let is block scoped. What that means is that a variable created with the let keyword is available inside the "block" that it was created in as well as any nested blocks. When I say "block", I mean anything surrounded by a curly brace {} like in a for loop or an if statement.

Eg.

```
function discountPrices (prices, discount) {
  var discounted = []


  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3  console.log(discountedPrice) // 150  console.log(finalPrice) // 150
  return discounted
}
```

Remember that we were able to log i, discountedPrice, and finalPrice outside of the for loop since they were declared with var and var is function scoped. But now, what happens if we change those var declarations to use let and try to run it?

```
function discountPrices (prices, discount) {
  let discounted = []

  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i)  console.log(discountedPrice)  console.log(finalPrice)
  return discounted
}

discountPrices([100, 200, 300], .5) //  ReferenceError: i is not defined
```

We get ReferenceError: i is not defined. What this tells us is that variables declared with let are block scoped, not function scoped. So trying to access i (or discountedPrice or finalPrice) outside of the “block” they were declared in is going to give us a reference error as we just barely saw.

**var** vs **let**

The next difference has to do with Hoisting. Earlier we said that the definition of hoisting was "The JavaScript interpreter will assign variable declarations a default value of undefined during what's called the 'Creation' phase." We even saw this in action by logging a variable before it was declared (you get undefined)

```
function discountPrices (prices, discount) {
  console.log(discounted) // undefined
  var discounted = []


  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }
  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150


  return discounted
}
```

I can't think of any use case where you'd actually want to access a variable before it was declared. It seems like throwing a ReferenceError would be a better default than returning undefined. In fact, this is exactly what let does. If you try to access a variable declared with let before it's declared, instead of getting undefined (like with those variables declared with var), you'll get a ReferenceError.

```
function discountPrices (prices, discount) {
  console.log(discounted) // ReferenceError
  let discounted = []
  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}
```

**let** vs **const**

Now that you understand the difference between var and let, what about const? Turns out, const is almost exactly the same as let. However, the only difference is that once you've assigned a value to a variable using const, you can't reassign it to a new value.

```
let name = 'akki'
const handle = 'akkis'

name = 'akkis' //
handle = '@akkis' //  TypeError: Assignment to constant variable.
```

The take away above is that variables declared with let can be re-assigned, but variables declared with const can't be.

Cool, so anytime you want a variable to be immutable, you can declare it with const. Well, not quite. Just because a variable is declared with const doesn't mean it's immutable, all it means is the value can't be re-assigned. Here's a good example.

```
const person = {
  name: 'akshay'
}

person.name = 'akshay kumar'

person = {} // Assignment to constant variable.
```

Notice that changing a property on an object isn't reassigning it, so even though an object is declared with const, that doesn't mean you can't mutate any of its properties. It only means you can't reassign it to a new value.

**Question 3 : Write a brief intro on available data types in Javascript**

Ans:-

**Data Types in JavaScript**

Data types basically specify what kind of data can be stored and manipulated within a program.

There are six basic data types in JavaScript which can be divided into three main categories:

**primitive** (or primary), **composite** (or reference), and **special data types**.

String, Number, and Boolean are primitive data types.

Object, Array, and Function (which are all types of objects) are composite data types.

Whereas Undefined and Null are special data types.

**Primitive data types** can hold only one value at a time, whereas composite data types can hold collections of values and more complex entities.

Let's discuss each one of them in detail.

**The String Data Type**

The string data type is used to represent textual data (i.e. sequences of characters). Strings are created using single or double quotes surrounding one or more characters, as shown below:

Eg.

```
var text1 = 'Hello…!';  // using single quotes
var text2 = "Hello there!";  // using double quotes
```

You can include quotes inside the string as long as they don't match the enclosing quotes.

Eg.

```
var a = "Let's have a cup of coffee."; // single quote inside double quotes
var b = 'He said "Hello" and left.';  // double quotes inside single quotes
var c = 'We\'ll never give up.';     // escaping single quote with backslash
```

**The Number Data Type**

The number data type is used to represent positive or negative numbers with or without decimal place, or numbers written using exponential notation e.g. 1.5e-4 (equivalent to 1.5x10-4).

Eg.

```
var a = 25;         // integer
var b = 80.5;       // floating-point number
var c = 4.25e+6;    // exponential notation, same as 4.25e6 or 4250000
var d = 4.25e-6;    // exponential notation, same as 0.00000425
```

The Number data type also includes some special values which are: Infinity, -Infinity and NaN. Infinity represents the mathematical Infinity ∞, which is greater than any number. Infinity is the result of dividing a nonzero number by 0, as demonstrated below:

Eg.

```
alert(16 / 0);  // Output: Infinity
alert(-16 / 0); // Output: -Infinity
alert(16 / -0); // Output: -Infinity
```

While NaN represents a special Not-a-Number value. It is a result of an invalid or an undefined mathematical operation, like taking the square root of -1 or dividing 0 by 0, etc.

Eg.

```
alert("Some text" / 2);       // Output: NaN
alert("Some text" / 2 + 10);  // Output: NaN
alert(Math.sqrt(-1));         // Output: NaN
```

**The Boolean Data Type**

The Boolean data type can hold only two values: true or false. It is typically used to store values like yes (true) or no (false), on (true) or off (false), etc. as demonstrated below:

Eg.

```
var isReading = true;   // yes, I'm reading
var isSleeping = false; // no, I'm not sleeping
```

Boolean values also come as a result of comparisons in a program. The following example compares two variables and shows the result in an alert dialog box:

Eg.

```
var a = 2, b = 5, c = 10;

alert(b > a) // Output: true
alert(b > c) // Output: false
```

**The Undefined Data Type**

The undefined data type can only have one value-the special value undefined. If a variable has been declared, but has not been assigned a value, has the value undefined.

Eg.

```
var a;
var b = "Hello World!"

alert(a) // Output: undefined
alert(b) // Output: Hello World!
```

**The Null Data Type**

This is another special data type that can have only one value-the null value. A null value means that there is no value. It is not equivalent to an empty string ("") or 0, it is simply nothing.

A variable can be explicitly emptied of its current contents by assigning it the null value.

Eg.

```
var a = null;
alert(a); // Output: null

var b = "Hello World!"
alert(b); // Output: Hello World!

b = null;
alert(b) // Output: null
```

**The Object Data Type**

The object is a complex data type that allows you to store collections of data.

An object contains properties, defined as a key-value pair. A property key (name) is always a string, but the value can be any data type, like strings, numbers, booleans, or complex data types like arrays, function and other objects. You'll learn more about objects in upcoming chapters.

The following example will show you the simplest way to create an object in JavaScript.

Eg.

```
var emptyObject = {};
var person = {"name": "Clark", "surname": "Kent", "age": "36"};

// For better reading
var car = {
    "modal": "BMW X3",
    "color": "white",
    "doors": 5
}
```

You can omit the quotes around property name if the name is a valid JavaScript name. That means quotes are required around "first-name" but are optional around firstname. So the car object in the above example can also be written as:

Eg.

```
var car = {
    modal: "BMW X3",
    color: "white",
    doors: 5
}
```

**The Array Data Type**

An array is a type of object used for storing multiple values in single variable. Each value (also called an element) in an array has a numeric position, known as its index, and it may contain data of any data type-numbers, strings, booleans, functions, objects, and even other arrays. The array index starts from 0, so that the first array element is arr[0] not arr[1].

The simplest way to create an array is by specifying the array elements as a comma-separated list enclosed by square brackets, as shown in the example below:

Eg.

```
var colors = ["Red", "Yellow", "Green", "Orange"];
var cities = ["London", "Paris", "New York"];

alert(colors[0]);   // Output: Red
alert(cities[2]);   // Output: New York
```

**The Function Data Type**

The function is callable object that executes a block of code. Since functions are objects, so it is possible to assign them to variables, as shown in the example below:

Eg.

```
var greeting = function(){
    return "Hello World!";
}


// Check the type of greeting variable
alert(typeof greeting) // Output: function
alert(greeting());     // Output: Hello World!
```

In fact, functions can be used at any place any other value can be used. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to other functions, and functions can be returned from functions. Consider the following function:

Eg.

```
function createGreeting(name){
    return "Hello, " + name;
}
function displayGreeting(greetingFunction, userName){
    return greetingFunction(userName);
}


var result = displayGreeting(createGreeting, "Peter");
alert(result); // Output: Hello, Peter
```

**The typeof Operator**

The typeof operator can be used to find out what type of data a variable or operand contains. It can be used with or without parentheses (typeof(x) or typeof x).

The typeof operator is particularly useful in the situations when you need to process the values of different types differently, but you need to be very careful, because it may produce unexpected result in some cases, as demonstrated in the following example:

Eg.

```
// Numbers
typeof 15;  // Returns: "number"
typeof 42.7;  // Returns: "number"
typeof 2.5e-4;  // Returns: "number"
typeof Infinity;  // Returns: "number"
typeof NaN;  // Returns: "number". Despite being "Not-A-Number"

// Strings
typeof '';  // Returns: "string"
typeof 'hello';  // Returns: "string"
typeof '12';  // Returns: "string". Number within quotes is typeof string

// Booleans
typeof true;  // Returns: "boolean"
typeof false;  // Returns: "boolean"

// Undefined
typeof undefined;  // Returns: "undefined"
typeof undeclaredVariable; // Returns: "undefined"

// Null
typeof Null;  // Returns: "object"

// Objects
typeof {name: "John", age: 18};  // Returns: "object"

// Arrays
typeof [1, 2, 4];  // Returns: "object"

// Functions
typeof function(){};  // Returns: "function"
```

As you can clearly see in the above example when we test the null value using the typeof operator (line no-22), it returned "object" instead of "null".

This is a long-standing bug in JavaScript, but since lots of codes on the web written around this behavior, and thus fixing it would create a lot more problem, so idea of fixing this issue was rejected by the committee that design and maintains JavaScript.