CSE 220 Web Programming and Scripting

DR. RAJALAKSHMI S

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING
SRI RAMACHANDRA FACULTY OF ENGINEERING AND TECHNOLOGY

Module 1

Introduction to JavaScript


Overview
Introduction to JavaScript
JavaScript Types
JavaScript Variables
JavaScript Values
JavaScript Objects
JavaScript Functions
JavaScript Operators
JavaScript Expressions
JavaScript Statements
JavaScript Scopes
JavaScript Looping Statement
JavaScript Conditional Statement
Sign Off
PREPARED BY DR.RAJALAKSHMI S
3


JS
Introduction
JavaScript is a dynamically typed language which means that variables can hold values of any type and their types can change during run time
PREPARED BY DR.RAJALAKSHMI S
4

JavaScript Types

Primitive Types

- String
- Boolean
- Number
- Null
- Undefined
- Symbol

Reference Types

- Object
- Array
- Function
- Date
- RegExp
- Error

PREPARED BY DR.RAJALAKSHMI S 5

Primitive Types

There are six primitive data types namely: String, Boolean, Number, Null, Undefined and Symbol. Primitive types are immutable, meaning their values cannot be changed once they are assigned.

String – Represents a sequence of characters enclosed in single or double quotes.

Ex: "Hello World!" , "JavaScript"

Number – Represents numeric values, including integers and floating – point numbers.

Ex: 42, 3.14

Boolean: Represents the logical values of true or false. Used for conditional statements and logical operations

Null: Represents the intentional absence of any object value. It is a special value assigned to variable when they are explicitly set to nothing or empty

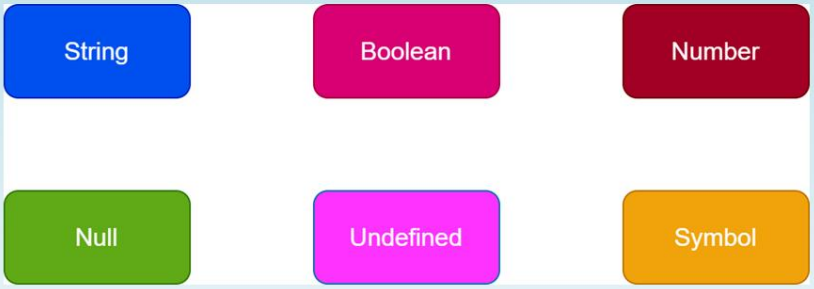
Undefined: Represents a variable that has been declared but not assigned a value. It is the default value of uninitialized variables

Symbol: Represents a unique and immutable value that can be used as an identifier for object properties. (Introduced in ECMAScript 2015)

PREPARED BY DR.RAJALAKSHMI S 6

Primitive Types

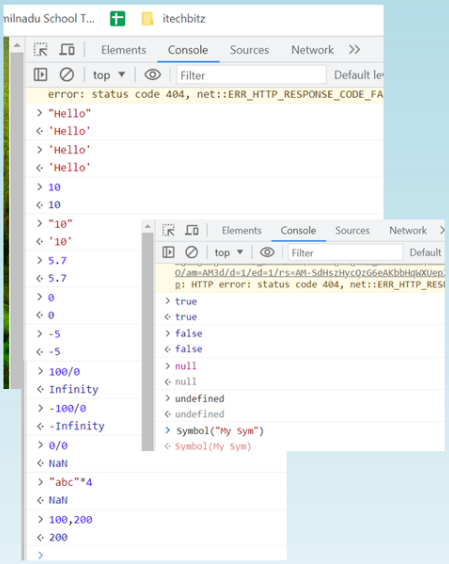
These primitive types are passed by value when assigned to variables or passed as arguments to functions. They do not have properties or methods of their own. However, JavaScript automatically converts them to their corresponding object wrappers (Number, String, Boolean) when necessary, allowing access to properties and methods.



PREPARED BY DR.RAJALAKSHMI S 7

Primitive Types - Example

```
let num = 52; // Number
let str = " Hello World"; // String
let bool = true; // Boolean
let undef = undefined; // Undefined
let n = null; // Null
let sym = Symbol ("Hello"); // Symbol

console.log(typeof num); // Output: "number"
console.log(typeof str); // Output: "string"
console.log(typeof bool); // Output: "boolean"
console.log(typeof undef); // Output: "undefined"
console.log(typeof n); // Output: "object" (typeof null is an anomaly in Javascript)
console.log(typeof sym);
```

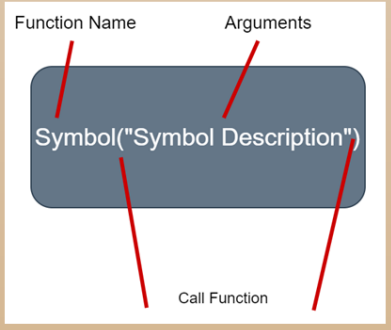


PREPARED BY DR.RAJALAKSHMI S 8

Primitive Types – Good to Know

- Numbers in Javascript don't divide into different subtypes like integer, decimal or something.
- Comma in Javascript is for separation of expression
- Null is an indicator of absence of any value
  - If specific variable should not have any value, you can assign value "null" to it as indicator of absence of any value.
  - But never assign "undefined" value for the same purpose
- Do not assign undefined to any value. Because undefined is used internally by JavaScript engine
- Symbols are used as property names for objects and there is default built in symbols
- Symbol values are all unique. Consider, I am comparing the value to itself, but still I get false

```
> Symbol("My Sym") == Symbol("My Sym")
< false
```



PREPARED BY DR.RAJALAKSHMI S 9

Reference Types

Objects that are stored and accessed by reference. Reference types allow for more complex data structures and behaviors. The basic reference types in JavaScript are:

Object: The most fundamental reference type in JavaScript. Objects are collections of key – value pairs and can store various types of data and functions. They can be created using object literals or constructor functions

Array: A special type of object that represents an ordered collection of elements. Arrays can store multiple values of different types and are accessed using numeric indices. They have built – in methods for manipulating and iterating over their elements

Function: Functions in JavaScript are first-class objects and can be assigned to variables, passed as arguments and returned as values. They allow for the creation of reusable blocks of code and support higher – order programming paradigms

PREPARED BY DR.RAJALAKSHMI S 10

Reference Types

Date: Represents a specific date and time. The Date object provides methods for creating, manipulating and formatting dates and times

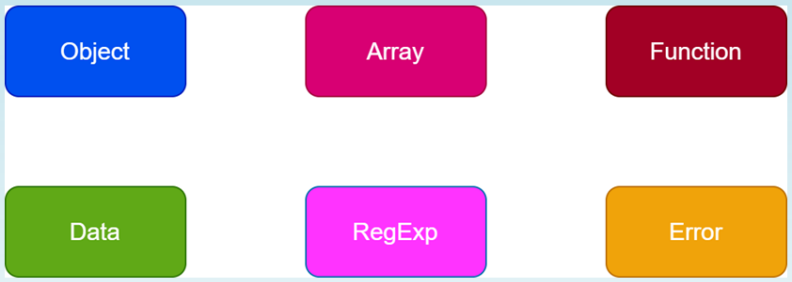
RegExp: Represents a regular expression, which is a pattern used to match and manipulate text. Regular expressions provide powerful string matching and pattern matching capabilities

Error: Represents an error object thrown during the execution of code. Error objects provide information about the type and cause of an error and can be used for error handling and debugging

PREPARED BY DR.RAJALAKSHMI S 11

Reference Types

- These reference types are mutable, meaning their values can be modified. They are accessed and manipulated by reference, as opposed to primitive types that are passed by value.
- Reference types is essential for working with more complex data structures and utilizing the built – in functionality provided by JavaScript



PREPARED BY DR.RAJALAKSHMI S 12

Primitive Types Vs Reference Types

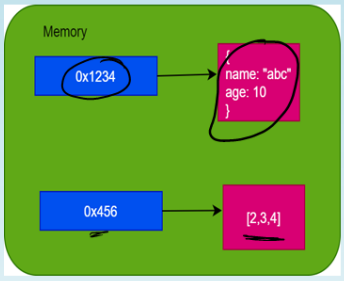
Primitive Type	Reference Type
Immutable: Primitive values are immutable, which means that once they have been assigned a value, it cannot be modified. Any procedure that alters a primitive value looks to modify it, but it actually creates a new value.	Mutable: Non-primitive values, such as objects and arrays, are modifiable. Even after they have been assigned to a variable, their values can still be changed.
Stored by value When a primitive value is assigned to a variable or passed as an argument, it is stored by value, which copies the original value to the new variable or function parameter. The original value is unaffected by modifications made to the new variable or parameter	Stored by reference: non-primitive values are stored by reference. A reference to the original value is generated when a non-primitive item is passed as an argument or assigned to a variable. The original value is impacted by modifications made to the new variable or parameter.
Primitive values are compared by value, meaning their values are compared directly.	Non primitive values are compared by reference, so two objects or arrays with the same properties and values are not considered equal unless they reference the same object in memory
Primitive types have a fixed size in memory	Non primitive types have a dynamic size depending on the content and structure of the object or array

PREPARED BY DR.RAJALAKSHMI S 13

Primitive Types Vs Reference Types

Storage of Primitive data type

In reference type, a pointer holds location of the object in memory

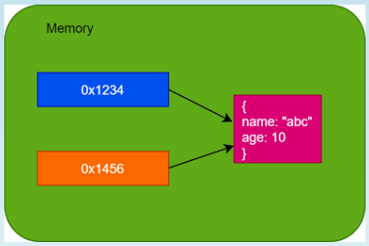


PREPARED BY DR.RAJALAKSHMI S 14

Reference Types

Is it possible for two different pointers to point to the same value in memory:

Yes. Two pointers may point ot the same value in memory. But pointers themselves are stored in memory.

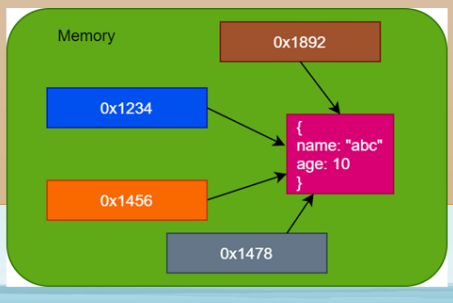


PREPARED BY DR.RAJALAKSHMI S 15

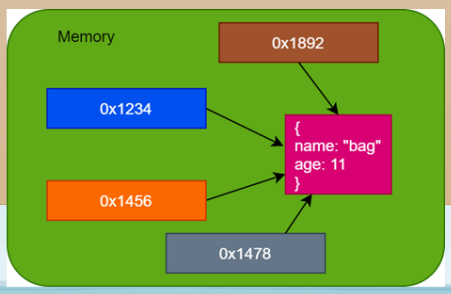
Primitive Types Vs Reference Types

There can be more than two pointers pointing to the same location in memory. And also it is possible to change (update) value of the object without disturbing the pointers

More than two pointers pointing to the same location



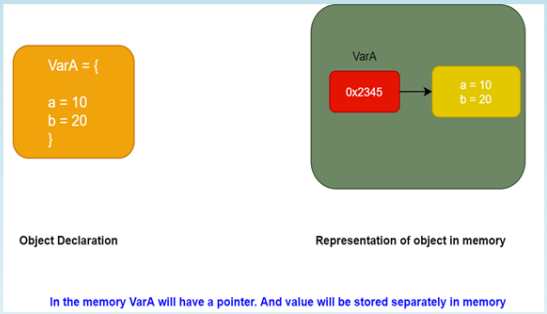
Updating the value



PREPARED BY DR.RAJALAKSHMI S 16

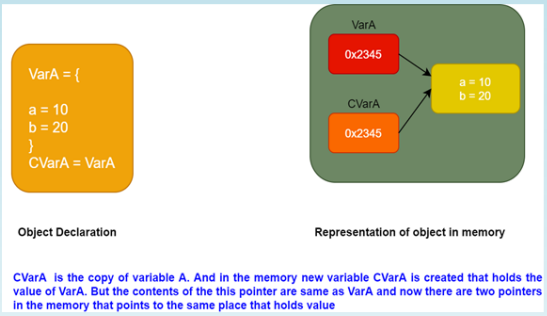
Reference Types - Example

Consider an object VarA holds the value for a = 10 and b = 25



In the memory VarA will have a pointer. And value will be stored separately in memory

Let's create a copy of VarA

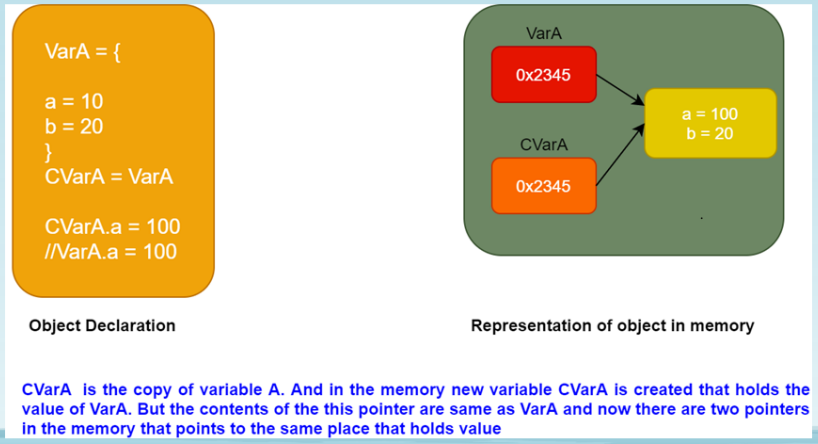


CVarA is the copy of variable A. And in the memory new variable CVarA is created that holds the value of VarA. But the contents of the this pointer are same as VarA and now there are two pointers in the memory that points to the same place that holds value

PREPARED BY DR.RAJALAKSHMI S 17

Reference Types - Example

Now let we update the value in the memory,



CVarA is the copy of variable A. And in the memory new variable CVarA is created that holds the value of VarA. But the contents of the this pointer are same as VarA and now there are two pointers in the memory that points to the same place that holds value

PREPARED BY DR.RAJALAKSHMI S 18

JavaScript Variables

Variables are building blocks of any programming language and variable is something that can change its value. Variables are containers that can hold value.

- **Purpose:** Variables holds a value and one can reuse variables as many times as needed. The value of the variable can be changed.
- In JavaScript Variables are used to store values and are declared using the 'var', 'let', or 'const' keywords.
- Variables in Javascript can hold different types of values including primitive values and references to non-primitive values.
- Each variable has name and process of variable creation is called variable declaration
- There are three reserved keywords used to declare a variable Var, Let and Const

PREPARED BY DR.RAJALAKSHMI S 19

Variable Declaration - Var

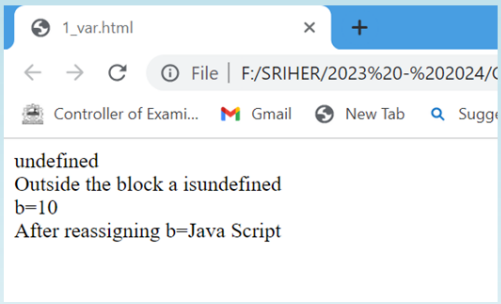
- variables declared with 'var' have function scope or global scope, meaning they are accessible within the function or globally throughout the program.
- They can be declared and assigned values multiple times within the same scope.
- Variables declared with 'var' are hoisted to the top of their scope, so they can be accessed before they are declared.
- JavaScript process all the declarations before it starts to execute the program. Thus, declaring the variable anywhere inside the code is equivalent to declaring it at the top of the code. Therefore, JavaScript gives the ability to use the variable before their declarations in the code. This behavior of variables is termed as **var hoisting**

```
var v1;     // variable declaration
var v2 = "cat" // variable initialization
```

PREPARED BY DR.RAJALAKSHMI S 20

Variable Declaration – Var Example

```
<html>
  <body>
    <script>
    {
      var a;
      document.write(a+"</br>")
    }
    document.write("Outside the block a is" + a  + "</br>");
    var b = 10;
    document.write("b="+b+"</br>")
    b = "Java Script"
    document.write("After reassigning b="+b+"</br>")
    </script>
  </body>
</html>
```



PREPARED BY DR.RAJALAKSHMI S 21

Variable Declaration - let

let introduces block scoping, which means that variables declared with 'let' are limited to the block (enclosed within curly braces{ }) in which they are defined including any nested blocks

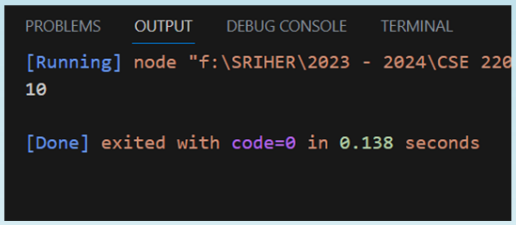
Key features:

- **Block scope:** variables declared with 'let' have block scope, which means they are accessible only within the block in which they are declared and not outside of it
- **Hoisting:** let variables are hoisted to the top of their scope, but they are not initialized until the actual declaration. This means that trying to access a 'let' variable before its declaration will result in Reference Error
- **Redefinition:** It is not possible to declare the same variable within the same scope using let. This leads to syntax error

PREPARED BY DR.RAJALAKSHMI S 22

Variable Declaration – let Example

```
if (true) {
  // console.log(x); // error ReferenceError: Cannot access 'x' before
initialization
  let x = 10;
  console.log(x);
}
// console.log(x); // ReferenceError: x is not defined
```



PREPARED BY DR.RAJALAKSHMI S 23

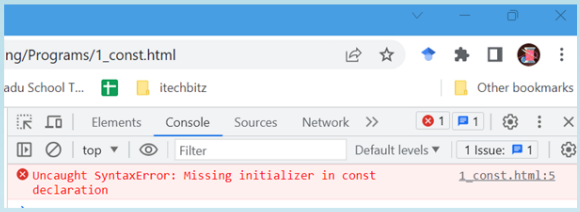
Variable Declaration - const

- Introduced in ECMAScript 2015 (ES6)
- 'const' is used to declare constants.
- Variables declared with 'const' have block scope, similar to 'let'
- 'const' variables are read – only and cannot be reassigned once they are assigned a value
- 'const' variables are not hoisted to the top of their scope.
- An initializer for a constant is required; i.e., one must specify its value in the same statement in which it is declared.

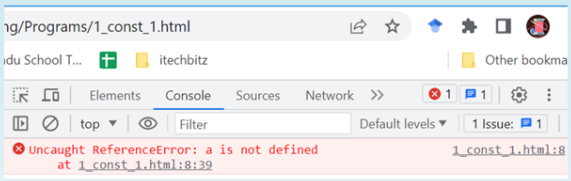
PREPARED BY DR.RAJALAKSHMI S 24

Variable Declaration – const Example

```
<html>
  <body>
    <script>
    {
      const a;
      document.write(a+"</br>")
    }
    </script>
  </body>
</html>
```




```
<html>
  <body>
    <script>
    {
      const a = 10;
      document.write("Inside block a = "+a+"</br>")
    }
    document.write("Outside block a = "+a+"</br>")
    </script>
  </body>
</html>
```



Variable Declaration – Good to Know

Choosing the appropriate keyword ('var', 'let' or 'const') depends on the desired scope and mutability of the variable.

Keyword	Scope	Hoisting	Redeclaration	Reassignment
'var'	Function	Yes	Allowed	Allowed
'let'	Block	No	Not Allowed	Allowed
'const'	Block	no	Not Allowed	Not Allowed

JavaScript Variables - Characteristics

JavaScript is Loosely typed

JavaScript is known as untyped/loosely/weakly typed language. This means that we do not have to specify the data type in advance unlike other languages.

```
var  x;
x = 5;
```

In the variable declaration, we no need to specify the data type for x

Javascript is dynamically typed

JavaScript is dynamically typed language meaning that once a variable is created with var, any kind of value can be stored in it.

```
var  x = 5; // x is assigned with number value
x = "JavaScript"; // x is assigned with string
```

PREPARED BY DR.RAJALAKSHMI S 27

JavaScript Variables - Characteristics

In Java Script, you can reassign value of any type to the existing variable declared with "let", "var".

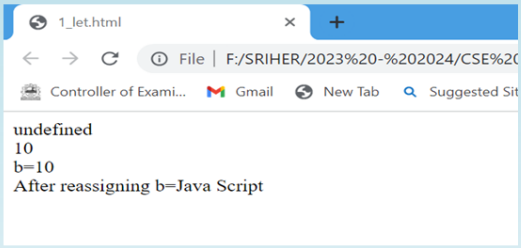
let

- Introduced in ECMAScript 2015 (ES6), 'let' allow for block – level scoping
- Variables declared with 'let' have block scope, meaning they are accessible only within the block in which they are defined.
- They can be reassigned but cannot be redeclared within the same block scope
- 'let' variables are not hoisted to the top of their scope, so they cannot be accessed before they are declared

PREPARED BY DR.RAJALAKSHMI S 28

JavaScript Variables - Characteristics Example

```
<html>
  <body>
    <script>
    {
      let a;
      document.write(a+"</br>")
      a = 10; // let a = 10 leads to error
      document.write(a+"</br>")
    }
    let b = 10;
    document.write("b="+b+"</br>")
    b = "Java Script"
    document.write("After reassigning b="+b+"</br>")
    </script>
  </body>
</html>
```

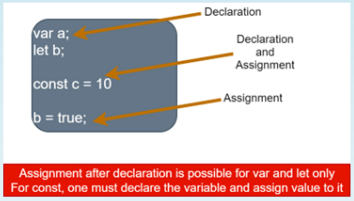


JavaScript Variables - Characteristics

With var and let, one can create variable and later on assign. But for const, variable declaration and assignment should happen in one line.

```
let a = 10;
a = 20;
var b = true
b = false
const c = 100;
c =200; // Type error assignment to constance
```

If try to access variable that is declared later using "let" or "const", then reference error: Variable is not defined arises. When variable is declared using "var" or "let" its value will be automatically set to undefined.



Variable declaration is a statement and variable assignment is an expression

For let and var variables, variables can be re initialized, whereas for const it is not possible

```
console.log(a) // Reference error a is not defined
let a; // Declaration
console.log(a); // Undefined
a = true;  // Assignment
console.log(a); // true
```

JavaScript Variables - **Static typed vs. dynamic typed**

Static typed: Type of each variable is defined before code is executed

Dynamic typed: Type of the variable will be set dynamically depending on the value that we assigned to the variable. In other words, the type of the variable is set dynamically during code execution. And this type can change during code execution.

Static Typing	Dynamic Typing
string a = "abc"; int b = 100; b = "xyz";	a = "abc"; b = 100; b = "xyz"

PREPARED BY DR.RAJALAKSHMI S 31

JavaScript Variables – Dynamic Typing

Java Script is dynamically typed language,

- This is advantage, as it is easy to assign values to the variables and type of the variable is set during the code execution.
- This is drawback, for example: if a variable "num" is assigned with a number, then accidentally if the value is changed to string (because this is possible in JavaScript) and if pass the variable num to a function which is expecting number, will throw error and it will be difficult to find where it is reassigned if the source code is large.

Possible solution:

Use const, because variable initial declaration and assignment happens at one line and re-assignment is not possible.

PREPARED BY DR.RAJALAKSHMI S 32

JavaScript Variables – Dynamic Typing

Dynamic typing is JS,

Consider the below example where initially a is assigned with 100 (a number), then true (a boolean) then Java Script (a string) and finally to object. It will not produce any error

```
let a = 100;

a = true;

a = "Java Script"

a = {};
```

```
function fun1(){
console.log("Hello")
}

fun1(); // Will display Hello

fun1 = 10;

fun1(); // Uncaught TypeError: fun1
is not a function
```

Hoisting is a behavior in Java Script where variable and function declarations are moved to the top of their respective scope during the compilation phase before the code is executed i.e., the variables and functions can be used before they are actually declared in the code. Only the declarations are hoisted not the initializations or assignments.

PREPARED BY DR.RAJALAKSHMI S 33

JavaScript Variables – Dynamic Typed - Example

The variable x is hoisted to the top of its scope, and the console.log statement runs without any error. However, x is undefined at that point because the initialization var x = 10; is not hoisted, only the declaration var x; is hoisted.

```
console.log(x);   // Undefined
var x = 10;
```

PREPARED BY DR.RAJALAKSHMI S 34

JavaScript Value

It is simply record in the memory and each record has specific type and all value types in JavaScript are divided into two groups:

Primitive Value Types	Reference Value Types
• String	• Object
• Boolean	• Array
• Number	• Function
• Null	• Date
• Undefined	• RegExp
• Symbol	• Error

Primitive Values

Number: Represents numeric values, such as 10, 10.12

String: Represents a sequence of characters, enclosed in single or double quotes, such as “Hello” or “JavaScript”

Boolean: Represents logical values ‘true’ or ‘false’

Undefined: Represents a variable that has been declared but not assigned a value

Null: Represents the intentional absence of any object value

Symbol: Represents a unique and immutable value used as an identifier

```
let age = 52; // Number
let name = "abc"; // String
let isWorking = true; // Boolean
let data = undefined; // Undefined
let n = null; // Null
let id = Symbol("Unique"); // Symbol
```

Non-Primitive Values (reference)

Object: Represents a collection of key – value pairs, created using object literals '{}' or constructor functions

Array: Represents an ordered collection of elements, created using square brackets '[ ]'

Function: Represents a block of reusable code that can be invoked by its name

Date: Represents a specific date and time

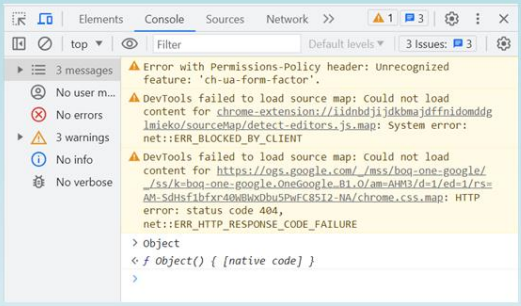
RegExp: Represents a regular expression for matching patterns in strings

Error: Represents an error object thrown during the execution of code

```
22  let person = {name:"ABC", age:25}; // object
23  let marks = [50, 76, 90]; // Array
24  let gr = function() {console.log{"Hello!"};}; //Function
25  let today = new Date(); // Date
26  let pattern = /abc/;
27  let error = new Error("No Internet"); // Error
```

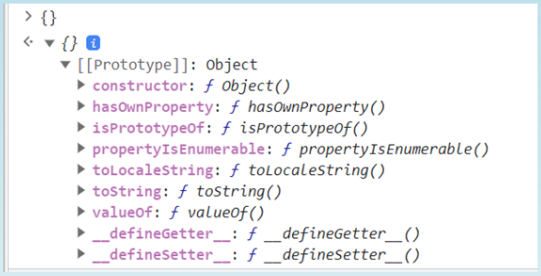

JavaScript Objects

- An object is a collection of keys – value pairs, where each key represents a property, and each value represents the corresponding data associated with that property.
- A property key can only be a string or a symbol
- Method is a property that has function as a value
- Object is a function
  - In Java Script there is a default variable called Object and this variable is a function and each function in JavaScript is Object



JavaScript Objects

- Objects in JavaScript are dynamic and flexible, allowing to add, modify or remove properties and their values at run time
- New empty object is an instance of Object
- Objects are a fundamental data type that allows you to store and organize collections of key – value pairs. Objects are versatile and flexible allowing you to represent complex data structures and define custom behaviors.



PREPARED BY DR.RAJALAKSHMI S 39

JavaScript Objects - Overview

- Creating object literal notation
- Object Constructor
- Accessing Object Properties
- Modifying object properties
- Adding and Removing Object Properties

PREPARED BY DR.RAJALAKSHMI S 40

JavaScript Objects - Creating Objects Literal

Objects can be created using object literal notation, which involves defining key – value pairs within curly braces '{ }'

An object object_name is defined with key value pairs. Each key value pair is separated by commas and enclosed in curly braces { }.

```
const object_name = {
key1 : value1
key2: value2
}
```

```
let person = {
  fname: "ABC",
  age: 18,
  dept: "CSE",
};

console.log(person);
```

JavaScript Objects - Object Constructor

Objects can also be created using the 'Object' Constructor function.

```
let person1 = new Object();
person1.fname = "CAD";
person1.age = 19;
person1["dept"] = "CSE";

console.log(person1);
```

JavaScript Objects - Accessing Object Properties

Object Properties can be accessed using dot notation or bracket notation.

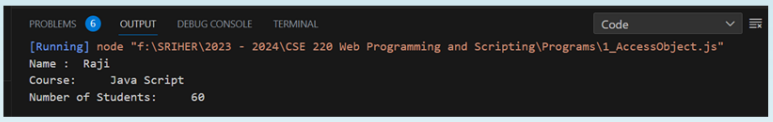
Dot notation

It is the most common way to access object the properties of objects.

It uses dot (.) followed by property name.

```
Object_name.property_name;
```

```
let person = {
  name: "Raji",
  course: "Java Script",
  num_Students: 60,
};

console.log("Name : \t" + person.name);
console.log("Course: \t" + person.course);
console.log("Number of Students: \t" +
person.num_Students);
```



PREPARED BY DR.RAJALAKSHMI S 43

JavaScript Objects - Bracket notation

- It involves using square brackets (‘[ ]’) to access object properties.
- Inside brackets **a variable, a string literal or an expression that evaluates to a string representing the property name.**

```
Object_name[property_name];
```

- Bracket notation allows to access object.
- It is useful when the property name is not known in advance or to access a property with a special character or a name that is not a valid identifier.

PREPARED BY DR.RAJALAKSHMI S 44

JavaScript Objects - Bracket notation Example

```
let person = {
  name: "Raji",
  age: 30,
  branch: "AIML",
  "staff-id": 12345,
};

const newAge = "age"; //using variable

const newbranch = "br" + "an" + "ch"; // expression

console.log("Name: \t" + person["name"]);
console.log("Age: \t" + person[newAge]);
console.log("Branch: \t" + person[newbranch]);
console.log("Staff ID: \t" + person["staff-id"]);

//console.log(person.staff - id); //ReferenceError: id is not defined
```

- In the code above, **'newAge'** variable contains the string **'age'** and **person[newAge]** uses bracket notation to access the **'age'** property of the person object
- Also, the **'newbranch'** variable is created by concatenating the string **"br" , "an" and "ch"** resulting in **'branch'**. Using *person[newbranch],* one can dynamically access the *"branch"* property of the person object.

```
PROBLEMS 6  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scriptin
Name:   Raji
Age:    30
Branch:     AIML
Staff ID:   12345

[Done] exited with code=0 in 0.159 seconds
```


JavaScript Objects - Modifying Object Properties

Object Properties can be modified by assigning a new value to them.

```
let student = {
  name: "ABC",
  age: 15,
  marks: [100, 99, 85, 92, 91],
  address: {
    city: "Chennai",
    state: "TamilNadu",
  },
};

console.log("Print Object");
console.log(student);

console.log("Mark1 of Student:\t" + student.marks[0]);

// change the value of mark1 to 78
student.marks[0] = 78; // same as student["marks"][0] = 78;

//change the age to 16
student.age = 16; // same as student["age"] = 16

//change the city to Trichy
student.address.city = "Trichy"; // student["address"]["city"] = "Trichy";

console.log("After Modification Print Object");
console.log(student);
```

Object_name[property_name] = newValue;
Object_name.property_name = newValue;

```
PROBLEMS 6  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\
Print Object
{
  name: 'ABC',
  age: 15,
  marks: [ 100, 99, 85, 92, 91 ],
  address: { city: 'Chennai', state: 'TamilNadu' }
}
Mark1 of Student:   100
After Modification Print Object
{
  name: 'ABC',
  age: 16,
  marks: [ 78, 99, 85, 92, 91 ],
  address: { city: 'Trichy', state: 'TamilNadu' }
}
```

JavaScript Objects - Bracket notation

- In the example above, the object student has the properties name, age, marks and address. The marks is an array containing multiple values. The property array can be accessed using the dot notation student.marks
- or bracket notation student[“marks”]. Both will give the array as the value of the property.
- To access specific elements within the array, one can use the index inside the square brackets. For example: **student.marks[0]** accesses the first element of the **marks** array, which is **100**. Similarly, student[‘marks’][1] accesses the second element 99.
- One can access the properties within an object and retrieve specific elements based on their index.

JavaScript Objects - Adding new Properties

A new property can be added to an object by simply assigning a value to a new key. The syntax is given as:

```
const person = {
  name: "ABC",
  age: 28,
};

person.address = "123 Main Road"; // using dot notation

person["gender"] = "Female"; // using bracket notation

person.hobby = ["gardening", "cycling"]; // adding array as property

person["hobby"][2] = "Reading"; // adding one more value for hobby array
property

const prop_name = "city"; // create a variable for city property
person[prop_name] = "Trichy"; // Dynamically adding the property
// person["hobby"][1] = "cooking"; // Modifying

console.log(person);
```

Object_name[new_property_name] = Value;
Object_name.new_property_name = Value;

```
PROBLEMS 6  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Script
{
  name: 'ABC',
  age: 28,
  address: '123 Main Road',
  gender: 'Female',
  hobby: [ 'gardening', 'cycling', 'Reading' ],
  city: 'Trichy'
}

[Done] exited with code=0 in 0.142 seconds
```

JavaScript Objects - Adding new Properties

- In this example, a new property called "address" to the "person" object is created by assigning the value '123 Main Road' to it.
- Similarly, a new property called "gender" to the "person" object is created using bracket notation and value of "Female" is assigned.
- A array property called hobby is created to include array of values. Later on, the third hobby of the person is included as person["hobby"][2] and a value of Reading is assigned.
- Also, a property can be added dynamically using bracket notation, especially when the property name is stored in a variable or needs to be generated programmatically. For example, prop_name variable is created to store the property "city" and then using bracket notation value is assigned to the property.
- Adding properties to an object is flexible in JavaScript that allows to expand and modify objects as needed during runtime.

JavaScript Objects - Removing Object Properties

Existing properties can be removed using the 'delete' operator.

```
const person = {
  name: "abc",
  age: 14,
  marks: {
    m1: 90,
    m2: 89,
    m3: 78,
  },
  subject: ["Tamil", "English", "Maths"],
  gender: "male",
};
console.log(person);

delete person.age; // removes the property age
console.log("After removing age");
console.log(person);

delete person.marks.m1; // Delete m1

delete person["marks"]["m2"]; // Delete m2
console.log("After removing mark");
console.log(person);

delete person.subject[0];

console.log("After removing subject");
console.log(person);
```

delete Object_name.property_name;

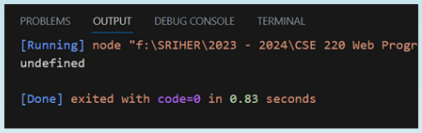
delete Object_name[property_name];

```
Module1 > JS 1_RemoveObjectProperty.js > ...

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
{
  name: 'abc',
  age: 14,
  marks: { m1: 90, m2: 89, m3: 78 },
  subject: [ 'Tamil', 'English', 'Maths' ],
  gender: 'male'
}
After removing age
{
  name: 'abc',
  marks: { m1: 90, m2: 89, m3: 78 },
  subject: [ 'Tamil', 'English', 'Maths' ],
  gender: 'male'
}
After removing mark
{
  name: 'abc',
  marks: { m3: 78 },
  subject: [ 'Tamil', 'English', 'Maths' ],
  gender: 'male'
}
After removing subject
{
  name: 'abc',
  marks: { m3: 78 },
  subject: [ <1 empty item>, 'English', 'Maths' ],
  gender: 'male'
```

JavaScript Objects - Reading the property that does not exist

Any attempt to read the property that does not exist results in undefined. For example, consider the object, obj which includes property pen and pencil with values 10 and 2 respectively. When tried to log the value for eraser it results in undefined as eraser is not the property of obj.

```
const obj = {
  pen: 10,
  pencil: 2,
};

console.log(obj.eraser);
```



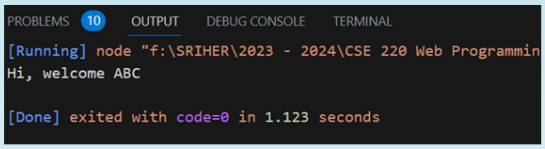
PREPARED BY DR.RAJALAKSHMI S 51

JavaScript Objects - Object Methods

Objects can also contain functions as property values, which are known as object methods. These methods can be invoked using dot notation and can operate on the object's properties.

In other words, Methods represent the properties with function as a property value.

```
let person = {
   name:"ABC",
   age: 25,
   greet:function(){
      console.log("Hi,  welcome " + this.name);
   }
};

person.greet(); // Output: Hi,  Welcome ABC
```



PREPARED BY DR.RAJALAKSHMI S 52

JavaScript Objects - Object Iteration

One can iterate over an object's properties using a "for... in" loop

```
let student = {
  name: "ABC",
  age: 15,
  marks: [100, 99, 85, 92, 91],
};

for (let key in student) {
  console.log(key + ":" + student[key]);
}
```

```
PROBLEMS 10  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Progra
name:ABC
age:15
marks:100,99,85,92,91

[Done] exited with code=0 in 0.129 seconds
```

PREPARED BY DR.RAJALAKSHMI S 53

JavaScript Objects - Nested Objects

An object may contain another object. Example consider the code below:

```
/**
 * Illustration of Nested Object
 */
let student = {
  name: "Raji",
  age: 18,
  dept: "AIML",
  mark: {
    Tamil: 96,
    Eng: 98,
  },
};

console.log("Print Object");
console.log(student);

console.log("Student Name: \t" + student.name);
console.log("Age: \t", student.age);
console.log("dept:" + student.dept);
console.log("Student Mark : \t" + student.mark);
console.log("Mark in tamil:\t" + student.mark.Tamil);
console.log("Mark in English:\t" + student.mark.Eng);
// Using bracket notation
// without "" inside bracket for property name displays ReferenceError: mark is
not defined
console.log("Mark in tamil:\t" + student["mark"]["Tamil"]);
```

```
// To modify the value
student.age = 15;

student.mark.Tamil = 91;
student["mark"]["Eng"] = 89;

console.log("After Modification");
console.log("Age: \t", student.age);
console.log("Student Mark : \t" + student.mark.Tamil);
console.log("Mark in tamil:\t" + student.mark.Eng);
```

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Script
Print Object
{ name: 'Raji', age: 18, dept: 'AIML', mark: { Tamil: 96, Eng: 98 } }
Student Name:   Raji
Age:    18
dept:AIML
Student Mark :  [object Object]
Mark in tamil:  96
Mark in English:    98
Mark in tamil:  96
After Modification
Age:    15
Student Mark :  91
Mark in tamil:  89
```

PREPARED BY DR.RAJALAKSHMI S 54

JavaScript Objects - Iteration in Nested Object

An object may contain another object. Example consider the code below:

```
// Example nested object
var data = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
    country: "USA",
  },
};
function iterateNestedObject(obj) {
  for (var key in obj) {
    if (obj.hasOwnProperty(key)) {
      if (typeof obj[key] === "object") {
        // If the value is an object, recursively call the function
        iterateNestedObject(obj[key]);
      } else {
        // Access the key-value pair
        console.log(key + ": " + obj[key]);
      }
    }
  }
}
// Call the function  to iterate over the nested object
iterateNestedObject(data);
```

```
PROBLEMS 10   OUTPUT   DEBUG CONSOLE   TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming
name: John
age: 30
street: 123 Main St
city: New York
country: USA

[Done] exited with code=0 in 0.136 seconds
```


JavaScript Objects - Const

If an object is declared as const in Java script, is it possible to change the value for property?

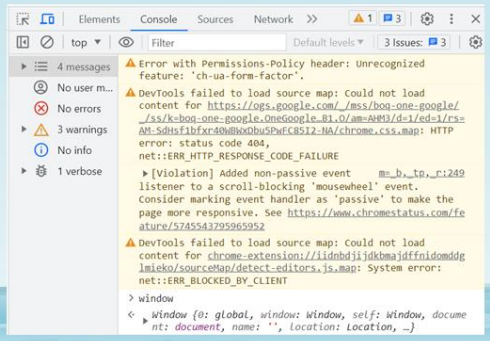
- If an object is declared as ‘const’ in JavaScript, one cannot reassign the entire object to a different value. ‘const’ keyword does not make objects or their properties immutable. It only means that the variable itself cannot be reassigned.
- While the object itself is not reassigned, one can still modify the properties of the object declared as ‘const’ as long as those properties are mutable.

```
const a = { age: 20, name: "Abc", dept: "cse" };

console.log(a);

// Adding properties
a.gender = "Male";
console.log("After adding property");
console.log(a);

//deleting property
delete a.dept;
console.log("After deleting property");
console.log(a);

//Assign an object with another value
a = { isWorking: true };
console.log("Not  possible for assignment");

console.log(a);
```

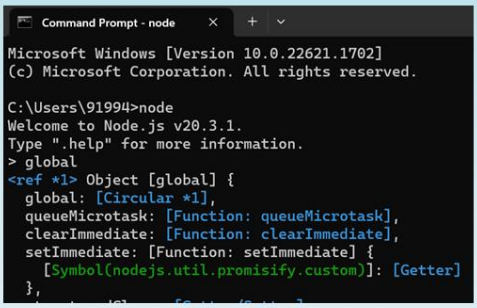
```
PROBLEMS 11   OUTPUT   DEBUG CONSOLE   TERMINAL                    Code
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\1_WorkingwithCon
{ age: 20, name: 'Abc', dept: 'cse' }
After adding property
{ age: 20, name: 'Abc', dept: 'cse', gender: 'Male' }
After deleting property
{ age: 20, name: 'Abc', gender: 'Male' }
f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\1_WorkingwithConstObject.js:16
a = { isWorking: true };
  ^

TypeError: Assignment to constant variable.
```

JavaScript Objects - Global Objects

There are two global objects Window and Global. Window is the global object in Web browser and Global is the global object in node.js. These objects contain list of properties, and some properties may contain nested Object.

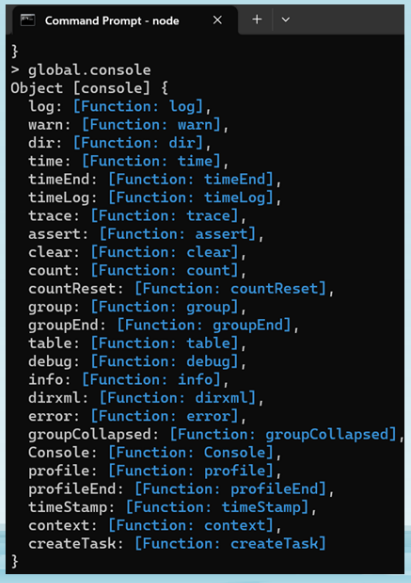


Console is one of the properties of Window. Open console and type node Then type global



JavaScript Objects - Console is also the property of global

```
> global.process.versions
{
  node: '20.3.1',
  acorn: '8.8.2',
  ada: '2.5.0',
  ares: '1.19.1',
  base64: '0.5.0',
  brotli: '1.0.9',
  cjs_module_lexer: '1.2.2',
  cldr: '43.0',
  icu: '73.1',
  llhttp: '8.1.1',
  modules: '115',
  napi: '9',
  nghttp2: '1.53.0',
  nghttp3: '0.7.0',
  ngtcp2: '0.8.1',
  openssl: '3.0.9+quic',
  simdutf: '3.2.12',
  tz: '2023c',
  undici: '5.22.1',
  unicode: '15.0',
  uv: '1.45.0',
  uvwasi: '0.0.18',
  v8: '11.3.244.8-node.9',
  zlib: '1.2.13.1-motley'
}
```



Console is also the property of global

```
> global.console.log
[Function: log]
>
```

```
> console.log("Hello Welcome!!!")
Hello Welcome!!!
undefined
>
```

As console is one of the property of global, the below way of accessing is syntactically meaningful.

```
> global.console.log("Hello Welcome!!!")
Hello Welcome!!!
undefined
```

JavaScript Objects - Global Objects

The document property is the key property of the window object. It refers to the Document object which represents the web page loaded in the current window or frame.

```
> window.document
<· ▶#document
>
```

Note:

- JavaScript Objects are dynamic, i.e., one can add, modify or remove properties at run time. They provide a powerful way to organize and manipulate data, and they serve as the building blocks for more complex data structures in Java Script.
- When the statements are executed in console browser, we will see undefined.
- Never manually assign undefined as a value of any variable or property of an object. Use "Null" instead
- Property is called Method if it holds function as a value.

JavaScript Functions

- A function contains blocks of code that need to be executed repeatedly or based on certain events.
- Why do we need Function? Function is reusable set of statements and expressions. In the example below, the same set of actions is repeated and again. To avoid repetitive blocks of code. Functions provide a way to organize and reuse code, making your programs modular and more manageable

```
let a = 10;
let b = 20;

c = a + b;

console.log("sum = " + c);

a = 100;
b = 200;

c = a + b;

console.log("sum = " + c);
```

```
PROBLEMS 10  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Prog
sum = 30
sum = 300

[Done] exited with code=0 in 0.146 seconds
```

JavaScript Functions

The above set of statements are replaced by function as shown below:

- The function declaration includes two variables d and f which serve as input parameters to the functions.
  The input to a function is passed through function call. The function can be called as many times as needed.
  The main advantage is that do not need to repeat the same blocks of code again and again.

```
let a = 10;
let b = 20;

function sum(d, f) {
  let e = d + f;
  console.log("sum = " + e);
}

sum(a, b);
a = 100;
b = 200;
sum(a, b);
```

```
PROBLEMS 10   OUTPUT   DEBUG CONSOLE   TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pr
sum = 30
sum = 300

[Done] exited with code=0 in 0.149 seconds
```


JavaScript Functions

Function can be,

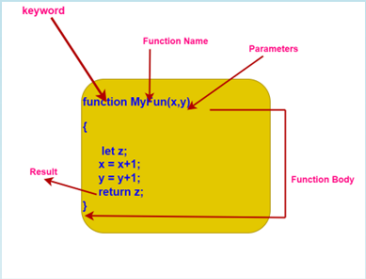
- a variable with a value,
- assigned as a value to the other function,
- anonymous
- used as argument to call to other function.

Function are categorized as,

1. Function definition
2. Function Invocation
3. Function Expression
4. Arrow functions

JavaScript Functions - Function definition

To define a function is by using the function keyword followed by function name, a set of parentheses for parameters (optional) and curly braces '{ }' to enclose the function body



Syntax

```
function fnname(p1,p2,...pn) {
}
```

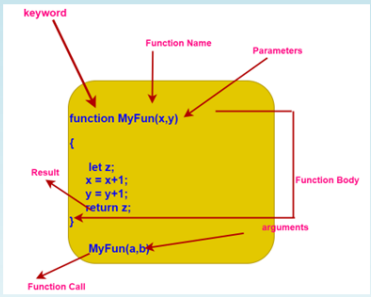
Example

```
function greet(name) {
  console.log('Hello, ' + name + '!');
}
```

In the example above, a function named "greet" is defined as a parameter 'name'. The function body logs a greeting message to the console using the provided name.

JavaScript Functions - Function Invocation

To execute a function, one needs to invoke or call it by using the function name followed by parentheses. Arguments can be passed into the function within the parentheses. Functions in JavaScript can also have return statements to send back a value as the result of function. Parameters allows to pass data into the function, and the function body can contain any valid JavaScript code.



Syntax

```
fname (argumentlist);
```

Example

```
greet("John");
```

JavaScript Functions - Function Expression

Function can also be defined using function expressions. In this case, the function is assigned to a variable property.

```
const square = function(num)
{
  return num *num;
}
```

A function expression is defined and assigned to the square variable. The function takes a parameter 'num' and returns the square of the provided number.

JavaScript Functions - Arrow functions

Arrow functions provide a concise syntax for writing functions, especially for simple, one line functions. They use => arrow syntax

```
const multiply = (a,b) => a*b;
```

In this example, an arrow function named 'multiply' takes two parameters and return their product.

JavaScript Functions - Function Declaration vs. Function Expression

In JavaScript one can use either function declaration or function expression. Function Expression are always anonymous.

	Function Declaration	Function Expression
Has name	Yes	No
Can be used standalone	Yes	No
Can be assigned to the variable	Yes	Yes
Can be passed as argument in the call to the other function	Yes	Yes

```
//Function Declaration
function myFn(a,b)
{
  let c;
  a  =  a+1;
  c = a +b;
  return c;
}
```

```
// Function Expression
function(a,b)
{
   let c;
   a = a + 1;
   c = a+ b;
   return c;
}
```

PREPARED BY DR.RAJALAKSHMI S 67

JavaScript Functions - Function Declaration

Function declarations are hoisted to the top of their scope, which means you can call the function before its actual declaration in the code. Function declarations are block-scoped to the nearest enclosing block. However, since they are hoisted, they can be called from anywhere within the current scope.

Syntax

```
function functionname(parameters)
{
}
```

Example

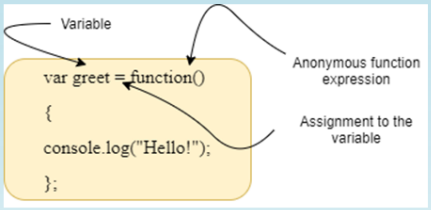
```
greet();   // Output: "Hello!"
function greet() {
  console.log("Hello!");
}
```

The function greet() is hoisted, so it can be called before its actual declaration in the code.

PREPARED BY DR.RAJALAKSHMI S 68

JavaScript Functions - Function Expression

Function expressions are not hoisted like function declarations. You need to declare the function expression before calling it. The scope of a function expression depends on how and where it is assigned.



Syntax

```
var functionName = function(parameters)
{
  // Function body
};
```

Example

```
greet(); // Error: greet is not a function
var greet = function() {
  console.log("Hello!");
};
```

In this example, the variable greet is hoisted, but since it's assigned a function expression, the assignment itself is not hoisted. As a result, greet is undefined when the function is called, causing an error.

JavaScript Functions - Function Expression

Function expression can be passed as an argument in function call. For example: consider the setTimeout(), a built in function in JavaScript which is used to schedule the execution of a function or an expression after a specified delay (in milli seconds). The function lets to introduce time intervals and delays in the code, making it useful for various asynchronous operations, animations and event handling scenarios.

Syntax

```
setTimeout(function, delay, arg1, arg2,...);
```

- 'function': the function or code snipped to be executed after the specified 'delay'
- 'delay': the time in milliseconds to wait before executing the function
- 'arg1, arg2,...': Optional argument list that needs to pass the function when it is called.

JavaScript Functions - Function Expression setTimeOut

setTimeout() is asynchronous. It means that it schedules the function to be executed in the future and continues executing the rest of the code without waiting for the delay to finish. This behavior is what allows non-blocking operations in JavaScript. If you need to cancel the execution of a scheduled function before it runs, you can use the value returned by setTimeout():

Function Declaration

```
function sayHello(name) {
  console.log(`Hello, ${name}!`);
}
setTimeout(sayHello, 3000, "Raji");
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 2
Hello, Raji!

[Done] exited with code=0 in 3.196 seconds
```

Function Expression

```
setTimeout(function ()
{
  console.log("Delayed Message");
}, 2000);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Progra
Delayed Message

[Done] exited with code=0 in 2.173 seconds
```


JavaScript Functions - Function Expression Callback

A callback function is a function that is passed as an argument to another function and is intended to be executed later, often after some asynchronous operation or event has been completed. Callback functions are commonly used in scenarios where we want to ensure that certain code runs only after a particular task has finished or a specific event has occurred. When you encounter the term "callback," it typically refers to the function that you pass to another function as an argument, which will be called later when some specific condition is met or when an asynchronous task is completed. The name "callback" is not a keyword in the language; it's just a naming convention to describe the function's purpose.

```
function greeting(name, callback) {
  console.log(`Hello, ${name}!`);
  callback();
}
function warmwish() {
  console.log("welcome!");
}
greeting("Raji", warmwish);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and
Hello, Raji!
welcome!

[Done] exited with code=0 in 0.135 seconds
```

Parameters, arguments and return statements are optional.

JavaScript Operators

Operators in JavaScript is simply a build in function. Various types of operators are as below,

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators (Relational Operators)
4. Logical Operators
5. Ternary Operators
6. Bitwise Operators
7. Unary Operators

JavaScript Operators - Arithmetic Operators

Arithmetic operators in JavaScript are used to perform mathematical calculations on numeric values.

Operator	Description	Example

+	Addition: Adds two or more values together	let a = 5;
		let b = 10;
		let sum = a + b;
		console.log(sum); // Output: 15

-	Subtraction: Subtracts the right operand from the left operand	let x = 10;
		let y = 3;
		let difference = x - y;
		console.log(difference); // Output: 7

*	Multiplication: Multiplies two or more values	let p = 4;
		let q = 6;
		let product = p * q;
		console.log(product); // Output: 24

/	Division: Divides the left operand by the right operand	let m = 12;
		let n = 3;
		let result = m / n;
		console.log(result); // Output: 4

%	Modulo: Returns the remainder after division	let num1 = 17;
		let num2 = 5;
		let remainder = num1 % num2;
		console.log(remainder); // Output: 2

++	Increment: Increases the value of a variable by 1	let count = 10;
		count++;
		console.log(count); // Output: 11

--	Decrement: Decreases the value of a variable by 1	let score = 8;
		score--;
		console.log(score); // Output: 7

Arithmetic operators in JavaScript are used to perform mathematical calculations on numeric values. The arithmetic operators work with numeric values, so if you try to use them with non-numeric values, JavaScript may try to implicitly convert the values to numbers before performing the operation, potentially leading to unexpected results.

JavaScript Operators - Assignment Operators

Assignment operators in JavaScript are used to assign values to variables. They allow to store data in variables and perform operations on the variable's current value.

Operator	Description	Example
=	Assignment: Assigns the value on the right to the variable on the left	let x = 10; let name = "John";

+=	Addition Assignment: Adds the value on the right to the current value of the variable on the left and assigns the result to the variable on the left	let a = 5;
		a += 2; // Equivalent to: a = a + 2;
		console.log(a); // Output: 7

-=	Subtraction Assignment: subtracts the value on the right from the current value of the variable on the left and assigns the result to the variable on the left	let b = 8;
		b -= 3; // Equivalent to: b = b - 3;
		console.log(b); // Output: 5

*=	Multiplication Assignment: Multiplies the current value of the variable on the left by the value on the right and assigns the result to the variable n the left	let c = 4;
		c *= 3; // Equivalent to: c = c * 3;
		console.log(c); // Output: 12

/=	Division Assignment: Divides the current value of the variable on the left by the value on the right and assigns the result to the variable on the left	let d = 15;
		d /= 5; // Equivalent to: d = d / 5;
		console.log(d); // Output: 3

%=	Modulo Assignment: Calculates the remainder when dividing the current value of the variable on the left by the value on the right and assigns the result to the variable on the left	let e = 17;
		e %= 5; // Equivalent to: e = e % 5;
		console.log(e); // Output: 2

JavaScript Operators - Comparison Operators (Relational Operators)

Comparison operators in JavaScript are used to compare values and return a Boolean result (true or false) based on the comparison's outcome. They are commonly used in conditional statements and loops to make decisions based on the values' relationships. Here are the main comparison operators in JavaScript:

Operator	Description	Example

==	Equal to: Check if two values are equal. It performs type coercion, meaning it can compare values of different data types after converting them to a common type	let x = 5;
		let y = "5";
		console.log(x == y); // Output: true (5 and "5" are considered equal after type coercion)

!=	Not equal to : check if two values are not equal. It also performs type coercion	let a = 10;
		let b = "10";
		console.log(a != b); // Output: false (10 and "10" are considered equal after type coercion)

===	Strict equal to: Checks if two values are equal without performing type coercion. It checks both values and data type	let p = 7;
		let q = "7";
		console.log(p === q); // Output: false (7 is a number and "7" is a string)

!==	Strict not equal to: Checks if two values are not equal without performing type coercion	let m = 12;
		let n = "12";
		console.log(m !== n); // Output: true (12 is a number and "12" is a string)

>	Greater than: Checks if the left operand is greater than right operand	let num1 = 5;
		let num2 = 3;
		console.log(num1 > num2); // Output: true

<	Less than: Checks if the left operand is less than the right operand.	let x = 8;
		let y = 10;
		console.log(x < y); // Output: true

>=	Greater than or equal to: Checks if the left operand is greater than or equal to the right operand.	let a = 7;
		let b = 7;
		console.log(a >= b); // Output: true

<=	Less than or equal to: Checks if the left operand is less than or equal to the right operand.	let p = 6;
		let q = 9;
		console.log(p <= q); // Output: true

JavaScript Operators - Logical Operators

Logical operators in JavaScript are used to combine or modify Boolean values and return a Boolean result (true or false) based on the evaluation of the operands. They are widely used to make decisions, perform conditional checks, and control the flow of programs. JavaScript has three main logical operators:

Operator	Description	Example

&&	Logical AND: Returns true if both operands are true; otherwise it returns false	let x = true;
		let y = false;
		console.log(x && y); // Output: false (Both x and y are not true)
		
		let x = 5;
		let y = 10;
		let z = 3;
		
		console.log(x < y && y < z); // Output: false (Both conditions are not true)
		console.log(x < y && y > z); // Output: true (Both conditions are true)

||	Logical OR: Returns true if at least one of the operands is true. Otherwise it returns false	let a = true;
		let b = false;
		console.log(a || b); // Output: true (At least one of a and b is true)
		
		let a = 5;
		let b = 10;
		let c = 3;
		
		console.log(a < b || b < c); // Output: true (At least one condition is true)
		console.log(a > b || b < c); // Output: false (Both conditions are false)

!	Logical NOT: Inverts the Boolean value of the operand. If the operand is true, it returns false; if the operand is false, it returns true.	let a = true; console.log(!a); // Output: false (Inverts the value of p, which is true)

Logical Operators can work with any Boolean expressions, not just variables. They can be used with the result of comparisons or logical operations

JavaScript Operators - Ternary Operators

Ternary operators take three operands and returns one value based on the evaluation of a condition, The syntax is given as:

condition ? expression1 : expression2

Condition: A condition that is evaluated to true or false

expression1: The value returned if the condition is true

expression2: The value returned if the condition is false

Operator	Description	Example

condition ? expression1 : expression2	Ternary operator works like if – else. If the condition is true, it evaluates to true else it evaluates to false	let age = 20;
		let result = age >= 18 ? "Major" : "Minor";
		console.log(result); // Output: "Major"

JavaScript Operators - Bitwise Operators

Bitwise operators operate on the individual bits of numeric values. Though these operators are not used in day-to-day programming, they can be used when working with binary data.

Operator	Description	Example

&	Bitwise AND: Performs a bitwise AND operation between each bit of two operands. It returns 1 in bit position in which the corresponding bits in both operands are 1	let num1 = 5; // Binary: 0101
		let num2 = 3; // Binary: 0011
		let result = num1 & num2; // Binary result: 0001 (Decimal: 1)
		console.log(result); // Output: 1

|	Bitwise OR: Performs a bitwise OR operation between each bit of the two operands. It returns 1 in bit position in which at least one of corresponding bits in operands are 1	let a = 6; // Binary: 0110
		let b = 3; // Binary: 0011
		let result = a | b; // Binary result: 0111 (Decimal: 7)
		console.log(result); // Output: 7

^	Bitwise XOR: Performs a bitwise exclusive OR (XOR) operation between each bit of the two operands	let x = 10; // Binary: 1010
		let y = 6; // Binary: 0110
		let result = x ^ y; // Binary result: 1100 (Decimal: 12)
		console.log(result); // Output: 12

~	Bitwise NOT: Performs a bitwise NOT operation on a single operand, inverting all the bits (0 becomes 1 and 1 becomes 0).	let num = 5; // Binary: 0101
		let result = ~num; // Binary result: 1010 (Decimal: -6)
		console.log(result); // Output: -6

<<	Left Shift: Shifts the bits of the left operand to the left by the number of positions specified by the right operand. The leftmost bits are filled with zeros	let num3 = 5; // Binary: 0101
		let result = num3 << 2; // Binary result: 010100 (Decimal: 20)
		console.log(result); // Output: 20

>>	Right Shift: Shifts the bits of the left operand to the right by the number of positions specified by the right operand The rightmost bits are filled based on the sign bit (0 for positive numbers and 1 for negative numbers)	let num4 = -15; // Binary: 11111111111111111111111111110001 (32-bit signed integer representation)
		let result = num4 >> 2; // Binary result: 11111111111111111111111111111100 (Decimal: -4)
		console.log(result); // Output: -4

PREPARED BY DR.RAJALAKSHMI S 79

JavaScript Operators - Unary Operators

Unary Operators work with a single operand, performing an operation on it or manipulating its value. The unary operators are listed as:

Operator	Description	Example

+	Unary Plus: Converts its operand into a number. If the operand is not already a number, JavaScript tries to convert it.	let num1 = "10";
		let num2 = +"5";
		console.log(typeof num1); // Output: string
		console.log(typeof num2); // Output: number

-	Unary Negation: Converts its operand into a number and then negates it (multiply by -1)	let a = -5;
		let b = -"3";
		console.log(a); // Output: -5
		console.log(b); // Output: -3

!	Logical NOT: Negates the Boolean value of its operands. If the operand is true, it returns false; otherwise, it returns true	let isTrue = true;
		let isFalse = !isTrue;
		console.log(isFalse); // Output: false

~	Bitwise Not: Inverts the bits of its operand (all 1s become 0s and all 0s become 1s)	let num = 5; // Binary: 0101
		let result = ~num; // Binary result: 1010 (Decimal: -6)
		console.log(result); // Output: -6

++	Increment: Increases the value of its operand by 1. The increment operator can be used as a prefix (++i) or postfix	let count = 5;
		console.log(++count); // Output: 6 (prefix increment)
		console.log(count++); // Output: 6 (postfix increment)
		console.log(count); // Output: 7

--	Decrease the value of its operand by 1. The decrement operator can also be used as a prefix or postfix.	let value = 10;
		console.log(--value); // Output: 9 (prefix decrement)
		console.log(value--); // Output: 9 (postfix decrement)
		console.log(value); // Output: 8

PREPARED BY DR.RAJALAKSHMI S 80

JavaScript Operators - Example

```
const prompt = require("prompt-sync")(); // required to get input from user
// Get the first number from the user
let firstNumber = prompt("Enter the first number:");

// Convert the input (which is a string) to a number
firstNumber = parseFloat(firstNumber);

// Get the second number from the user
let secondNumber = prompt("Enter the second number:");

// Convert the input (which is a string) to a number
secondNumber = parseFloat(secondNumber);

// Perform arithmetic operations
let sum = firstNumber + secondNumber;
let difference = firstNumber - secondNumber;
let product = firstNumber * secondNumber;
let quotient = firstNumber / secondNumber;

// Display the results
console.log("Sum: " + sum);
console.log("Difference: " + difference);
console.log("Product: " + product);
console.log("Quotient: " + quotient);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_Operators.js
Enter the first number:3
Enter the second number:4
Sum: 7
Difference: -1
Product: 12
Quotient: 0.75
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1>
```


JavaScript Expressions vs Statements

In JavaScript expressions and statements are two fundamental building blocks used to construct programs. They serve different purposes and have different characteristics.

Expressions

- An expression is any valid unit of code that evaluates to a value
- It can be as simple as a single value or as complex as a combination of values, variables and operators
- Expressions are often used to compute values or represent relationship between values

```
"dev"    //"abc"
20       // 20
Var1 + Var2  // sum of Var1 and Var2
"Good" + "day"  //Good day
X <= Y && !Z    // true or false
myFun(a,b)  // function result
```

Expressions can be used as arguments in function call.

JavaScript Expressions vs Statements

Statements

- A statement is a complete unit of code that performs an action or operation.
- Statements are executable and may produce side effects, change the program's state or control the flow of execution.
- A JavaScript program is essentially a sequence of statements executed one after another. The end of the statements are marked with semicolon.

```
let x = 5; // variable declaration statement
console.log("Hello, World"); // Function call statement
if(x > 0){  // Conditional Statement
// code block can contain multiple statements and expressions
console.log(x)
}
```

- Expressions can be turned into statements but vice versa is not possible.
- return is a statement not an expression

JavaScript Scopes

JavaScript has two main types of scopes: global scope and local (or function) scope. A scope in JavaScript determines the accessibility and visibility of variables and functions throughout the code.

They are grouped as,

1. Global Scope
2. Local Scope
3. Nested Scope
4. Block Scope

JavaScript Scoped - Global Scope

Variables and functions declared outside of any function or block have global scope. Global variables and functions can be accessed from anywhere in the code including inside functions or blocks. They are accessible throughout the entire program, from the point of declaration until the end of the program.

```
let gv = 5;

myFun();

function myFun() {
  console.log(gv);
}
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Prog
5

[Done] exited with code=0 in 1.392 seconds
```

PREPARED BY DR.RAJALAKSHMI S 85

JavaScript Scoped - Local Scope

Variables declared inside a function have local scope and are accessible only within that function. Function parameters also have local scope and are treated as local variables within the function. Local variables and function parameters are created each time the function is called and destroyed when the function completes execution.

```
function Myloc() {
  let var1 = "Hello";
  console.log(var1);
}

//console.log(var1); // ReferenceError: var1 is not defined
Myloc();
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 We
Hello

[Done] exited with code=0 in 0.127 seconds
```

PREPARED BY DR.RAJALAKSHMI S 86

JavaScript Scoped - Nested Scope

- When functions are defined inside other functions, they create nested scopes
- Inner functions have access to variables declared in their containing (outer) functions as well as global variables
- However, outer functions do not have access to variables declared in inner functions

```
function outfun() {
  var outvar1 = "I am variable in outer funtion";

  function infun() {
    var invar1 = "I am variable in outer funtion";
    console.log(outvar1); // can access here
    console.log(invar1); // can access here
  }
  infun();
  // console.log(invar); // ReferenceError: invar is not
defined
}

outfun();
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220
I am variable in outer funtion
I am variable in outer funtion

[Done] exited with code=0 in 0.143 seconds
```

PREPARED BY DR.RAJALAKSHMI S 87

JavaScript Scoped - Block Scope

Variables declared with 'let' and 'const' have block scope which means they are only accessible within the block in which they are declared (EX: if, for, while)

```
let a = 6;
if (a) {
  let b = 5;
  var c = 10;
  const d = 20;
  console.log("within  block");
  console.log("a=", a);
  console.log("b=", b);
  console.log("c=", c);
  console.log("d=", d);
}
console.log("Outside  block");
console.log("a=", a);
//console.log("b=", b);//ReferenceError: b is not defined
console.log("c=", c);
//console.log("d=", d); //ReferenceError: d is not defined
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
within block
a= 6
b= 5
c= 10
d= 20
Outside block
a= 6
c= 10
```

PREPARED BY DR.RAJALAKSHMI S 88

JavaScript Statements

The statements in JavaScript are categorized by loops and conditional as below,

Looping Statements

- For
- While
- Do... While

Conditional Statements

- If
- If...else
- if.. else if...else
- switch... case..
- break
- continue

PREPARED BY DR.RAJALAKSHMI S 89

JavaScript Statements – Loops - For

The for loop is a control flow statement in JavaScript that allows to execute a block of code repeatedly for a specific number of times or based on a defined condition. The for loop consists of three parts within its parentheses, separated by semicolons:

Initialization: The loop variable is initialized before the loop starts. It usually sets the initial value of a counter variable

Condition: The loop continuous executing as long as the condition is true. It is evaluated before each iteration of the loop.

Increment / Update: The loop variable is updated after each iteration. It typically increments or decrements the counter variable.

The general structure of for loop is:

```
for(Initial Statement; Condition; Iteration Action)
{
   //code
}
```

PREPARED BY DR.RAJALAKSHMI S 90

JavaScript Statements – Loops – For - Example

Loop to display iteration number from 0 to 5

```
for (let i = 1; i <= 5; i++) {
  console.log("I am iteration:", i);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
I am iteration: 1
I am iteration: 2
I am iteration: 3
I am iteration: 4
I am iteration: 5

[Done] exited with code=0 in 0.14 seconds
```

Loop to display from 5 to 1

```
for (let i = 5; i >= 1; i--) {
  console.log("I am iteration:", i);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONS
I am iteration: 5
I am iteration: 4
I am iteration: 3
I am iteration: 2
I am iteration: 1
```


JavaScript Statements – Loops – For - Example

Loop to display the elements in array

```
subjects = ["Tamil", "English", "Maths"];
for (let i = 0; i < subjects.length; i++) {
  console.log("Subject:[", i, "]:" + subjects[i]);
}
```

The **for** loop is used to iterate over the arrays and to execute the code for a specific number of times.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
Subject:[ 0 ]:Tamil
Subject:[ 1 ]:English
Subject:[ 2 ]:Maths

[Done] exited with code=0 in 0.144 seconds
```

JavaScript Statements – Loops - While

Another conditional control flow statement is while loop. It allows to execute the block of code repeatedly as long as specified condition is true. Unlike 'for' loop it does not have initialization or increment / update step. Rather it relies on condition to control the loop execution. The general syntax of while loop is as follows:

Working of while loop is as follows:

- The condition is evaluated before the loop starts. If the condition is true, the code inside the loop's block is executed
- After executing the block of code, the condition is evaluated again. If it is still true, the block is executed again. The process continues until the condition becomes false.
- If the condition is false while checking the condition for first time, the code inside the loop's block is never executed.

```
while(Condition)
{
   //code
}
```

PREPARED BY DR.RAJALAKSHMI S 93

JavaScript Statements – Loops – While - Example

Sum of elements in array

```
let a = [20, 10, 30, 50, 40];
let i = 0;
let sum = 0;
while (i < a.length) {
  sum = sum + a[i];
  i++;
}
console.log("sum=", sum);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
sum= 150

[Done] exited with code=0 in 0.133 seconds
```

Loop through an array and display its elements

```
let fruits = ["apple", "banana", "orange"];
let index = 0;
while (index < fruits.length) {
  console.log(fruits[index]);
  index++;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220
apple
banana
orange

[Done] exited with code=0 in 1.309 seconds
```

The while is useful when there is need to repeat code based on condition and don't have a specific number of iterations in advance. It is necessary to ensure to avoid infinite loops, make sure the condition eventually becomes false to exit the loop

PREPARED BY DR.RAJALAKSHMI S 94

JavaScript Statements – Loops – Do... While

The "do.. while" loop is a variation of the "while" loop. It allows to execute a block of code repeatedly as long as a specified condition is true. However, unlike the 'while' loop, the 'do..while' loop executes the code block at lease once before checking the condition. This means that the code inside the loop will always execute at least once regardless of whether the condition is true from the beginning. The syntax of do..while loop is as follows:

The working of do..while loop is as:

- The code inside the body of the loop will be executed first regardless of whether the condition is true or false
- After executing the block of statements once, the condition is evaluated
- If the condition is true, then the block will be executed again until the condition becomes false.

```
do
{
    //code
}
while(condition);
```

PREPARED BY DR.RAJALAKSHMI S 95

JavaScript Statements – Loops – Do...While - Example

Sum of numbers in an array

```
let a = [10, 20, 30];
let index = 0;
let sum = 0;
do {
  sum = sum + a[index];
  index++;
} while (index < a.length);
console.log("sum = ", sum);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Prog
sum =  60

[Done] exited with code=0 in 0.143 seconds
```

Factorial of a number using do..while loop

```
const prompt = require("prompt-sync")();

let n;
n = prompt("Enter the number");
n = parseInt(n);
let fact = 1;
let i = 1;

do {
  fact = fact * i; // fact *= i
  i++;
} while (i <= n);

console.log("Factorial = ", fact);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programmi
Enter the number5
Factorial =  120
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programmi
```

PREPARED BY DR.RAJALAKSHMI S 96

JavaScript Statements – Conditional - if

Conditional statements allow to execute different blocks of code based on specified conditions. They control the flow of the program, making decisions and taking actions based on whether certain conditions are true or false.

The 'if' statement allows to execute a block of code if a specified condition is true. If the condition is false, then the block of statement will not be executed. The syntax of if statement is given as:

```
if(condition)
{
   //code
}
```

The condition is specified inside the parenthesis followed by the keyword if. The condition inside the parenthesis will be evaluated which evaluates to a Boolean value either True or False. If the condition is true, the code inside the curly braces will be executed. If the condition is false, then the block of statements is not executed and the control will move to the next statement after if block.

JavaScript Statements – Conditional – If - Example

```
const prompt = require("prompt-sync")();

let n;
n = prompt("Enter the number");
n = parseInt(n);

if (n > 10) {
  console.log("The  number is greater than 10.");
}
```

This example, prompts the user to enter a number. If the entered number is less than 10, no action will be taken. If the entered number is greater than 10, then the message, "The number is greater than 10" will be displayed.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_if_1.js
Enter the number5
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_if_1.js
Enter the number11
The number is greater than 10.
```

JavaScript Statements – Conditional – if...else

The if...else statement is used to provide alternative code blocks to be executed based on a specific condition. If the condition on the 'if' statement is true, the code inside the 'if' block is executed. If the condition is false, the code inside the 'else' block is executed. The syntax of if...else is as follows:

- The condition is specified inside the parenthesis of if statement. It evaluates to Boolean value wither true or false
- If the condition is true, the code inside the curly braces {} will be executed and the 'else' block will be skipped
- If the condition is false, the code inside the 'else; block will be executed and the 'if' block will be skipped

```
if(condition)
{
    //code block 1
}
else
{
    //code block 2
}
```


JavaScript Statements – Conditional – If...else - Example

```
const prompt = require("prompt-sync")();

let age = prompt("Enter the age");
age = parseInt(age);

if (age >= 18) {
  console.log("Eligible  to vote");
} else {
  console.log("Not  Eligible  to vote");
}
```

In this example, if the value of age is greater than or equal to 18, the code inside the if statement will be executed. If the value of age is less than 18, then the code inside the else part will get executed.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripti
Enter the age19
Eligible to vote
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripti
Enter the age4
Not Eligible to vote
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripti
```

JavaScript Statements – Conditional – if...else if...else

The if...else if.. else statement is used in JavaScript to conditionally execute different blocks of code based on multiple specified conditions. It allows to check multiple conditions and execute different code blocks accordingly. The syntax of if..elseif..else is given as:

```
if(condition 1)
{
    //code block 1
}
else if(condition 2)
{
    //code block 2
}
else
{
    //code block 3
}
```

PREPARED BY DR.RAJALAKSHMI S 101

JavaScript Statements – Conditional – If...else If...else- Example

```
function Greetings() {
  var today = new Date();
  var hour = today.getHours();
  var minu = today.getMinutes();
  var sec = today.getSeconds();
  if (hour < 12) {
    console.log("GoodMorning  !!! Time is" + hour + ":" + minu + ":" +
sec);
  } else if (hour > 12 && hour < 16) {
    console.log("Good  Noon !!!TIme is" + hour + ":" + minu + ":" + sec);
  } else {
    console.log("Good  Day !!!TIme is" + hour + ":" + minu + ":" + sec);
  }
}

Greetings();
```

In this example, get the hour, minutes and seconds from date object. If the value is less than 12, it will print Good Morning with the current time. If the value is greater than 12 and less than 16, then it displays Good afternoon. Else, it will print Good Day.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 W
GoodMorning !!! Time is9:59:53

[Done] exited with code=0 in 0.136 seconds
```

PREPARED BY DR.RAJALAKSHMI S 102

JavaScript Statements – Conditional – switch… case..

The switch statement is another way to conditionally execute different blocks of code based on different cases or values of an expression. It is an alternative way of using if.. else if.. else statements when there is need to check a single expression against multiple possible values. The basic syntax of 'switch' statement is as follows:

```
switch(expression)
{
   case 1:
     //code
     break;
   case 2:
     //code
     break;
   ....
   default:
     //default block
     break;
}
```

PREPARED BY DR.RAJALAKSHMI S 103

JavaScript Statements – Conditional – If…else If…else- Example

```
const prompt = require("prompt-sync")();

let n = prompt("Enter a number");

n = parseInt(n);

switch (n) {
  case -2:
    console.log("The number is -2");
    break;
  case 0:
    console.log("The number is 0");
    break;
  case 2:
    console.log("The number is 2");
    break;
  default:
    console.log("Some  other number");
    break;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_switch_case.js
Enter a number2
The number is 2
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_switch_case.js
Enter a number0
The number is 0
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_switch_case.js
Enter a number-2
The number is -2
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_switch_case.js
Enter a number-10
Some other number
```

PREPARED BY DR.RAJALAKSHMI S 104

JavaScript Statements – Conditional – Break

The break statement is used to terminate the execution of a loop or switch statement. It is commonly used inside loops or switch cases to exit the loop or switch block prematurely when a certain condition is met. The syntax is:

```
break;
```

In loops:

When the break statement is encountered inside a loop, it immediately exits the loop and the program continues executing the code that follows the loop.

```
for (let i = 1; i < 10; i++) {
  if (i == 3) {
    console.log("I  ll stop the loop");
    break;
  }
  console.log(i);
}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
1
2
I ll stop the loop

[Done] exited with code=0 in 2.283 seconds
```


JavaScript Statements – Conditional – Switch Case

When the break statement is encountered inside a 'switch' case, it exits the 'switch' block preventing the execution of subsequent cases.

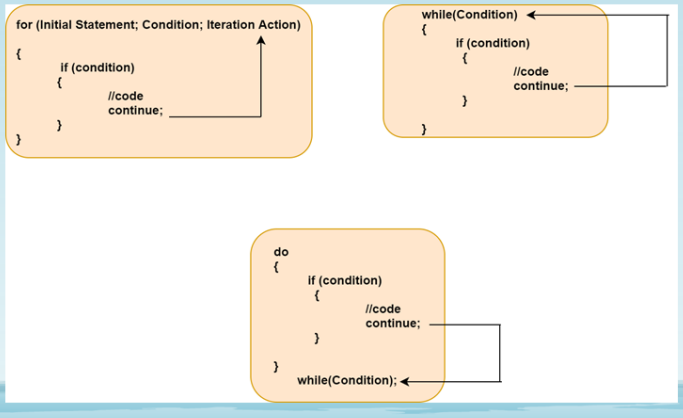
```
const prompt = require("prompt-sync")();
let month = prompt("Enter the month");
month = parseInt(month);

let year = prompt("Enter the year");
year = parseInt(year);

let daycount;

switch (month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    daycount = 31;
    console.log("Number  of days=" + daycount);
    break;
```

```
case 4:
  case 6:
  case 9:
  case 11:
    daycount = 30;
    console.log("Number  of days=" + daycount);
    break;
  case 2:
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
      daycount = 29;
      console.log("Number  of days=" + daycount);
    } else {
      daycount = 28;
      console.log("Number  of days=" + daycount);
    }
    break;
  default:
    console.log("Invalid  Month");
    break;
}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1> node 1_break_switch.js
Enter the month2
Enter the year2023
Number of days=28
PS F:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripting\Programs\Module1>
```

JavaScript Statements – Conditional – Continue

The continue statement is used inside loops (such as 'for', 'while' or 'do.. while') to skip the current iteration and proceed to the next iteration of the loop. When the 'continue' statement is encountered, the remaining code inside the loop for the current iteration is skipped, and the loop proceeds with the next iteration, if any.



PREPARED BY DR.RAJALAKSHMI S 107

JavaScript Statements – Conditional – Continue - Example

In this example, the if condition is true for all even numbers, thus continue statements take the control to the beginning of the loop making to print only odd numbers

```
for (let i = 0; i < 10; i++) {
  if (i % 2 == 0) {
    continue;
  }
  console.log(i);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pr
1
3
5
7
9

[Done] exited with code=0 in 0.141 seconds
```

In this example, divisors of 3 are skipped by using continue statement inside if condition.

```
let i = 0;
while (i < 10) {
  if (i % 3 == 0) {
    //console.log(i);
    i++;
    continue;
  }
  console.log(i);
  i++;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 W
1
2
4
5
7
8

[Done] exited with code=0 in 0.145 seconds
```

PREPARED BY DR.RAJALAKSHMI S 108

JavaScript Statements – Conditional – Continue - Example

In this example, the if condition is true for all even numbers, thus continue statements take the control to the beginning of the loop making to print only odd numbers

```
let i = 0;
do {
  if (i % 2 == 0) {
    i++;
    continue;
  }
  console.log(i);
  i++;
} while (i < 10);
```

Note:

The alert() function is available in Web browsers as part of the window object, but it is present in the Node.js environment or some other non-browser environments. Therefore, attempting to use alert() in a non-browser environment will result in a ReferenceError because the function is not defined in that context.

Reference

https://itechbitz.com/docs/programming-with-javascript__trashed/introduction-to-javascript/


Sign Off
PREPARED BY DR.RAJALAKSHMI S
110