function loopholes

(1)

```
var f12 = function fool2 () {

          }

X fool2(); // fool2 is not defined
  f12(); //
```

typeof f12
"fuction"

typeof fool2
↓
"undefined"

(2)

```
var f1 = function fool() {
           console.log("Hello");
         }

f1 //    f fool() {
           console.log("Hello");
         }
```

(3)

```
Var f2 = function() {
           console.log("Hello");
         }

f2 //     F() {
           console.log("Hello");
         }
```

→ function name not required but we can write

typeof f2
↓
"function"

(4)

```
function FF() { }
typeof FF // => "function"
```

→ Type coersion →

automatic type conversion

Ex-1 `123 + "4" → "1234"`

Number + String

String + String → String

Ex-2

```
var x = 10;
var y = 20;
var z = x + y;
```

10 + 20

Number (+) Number → addition operator

Number (30)

```
var x = "10";
var y = "20";
var z = x + y
```

String (+) Number → concatenation operator

String ("1020");

Ex-3

```
var x = 10;
var y = 20;
var z = "30";
var result = x + y + z;
```

30 (+) "30" → Concatenation

String "3030"

Number + Number → addition

10 (+) 20 → 30

→ JavaScript comes from the 'c' family of programming language. (like for loop, if-else, switch basic things are same)

→ What is JavaScript :-

- JavaScript often shortend as JS
- is a lightweight, interpreted, object oriented language with first-class functions,
- and is best known as the scripting language for web pages but it's used in many non-browser environment as well.

* Lightweight :- Small memory footprint, easy to implement

* Interpreted :- No compilation. Instructions executed directly.
  → No compilation means like in java or other languages first code will be compiled and a intermediate file will be generated (like bytecode in java or binary object file in C++ etc) but in case of JS is no intermediate file is generated directly instructions are executed one by one.

* Object-oriented :- modeled around object.

* First-class functions :- functions can be used as values
  → function can be a value, you can pass functions as parameter to method call.

* Scripting language :- is a basically a language where instructions or statements are written for a runtime environment.

~~[redacted]~~ you have commands that you can run on Runtime environment

Ex:
- ShellScript for Linux
- Power Shell for windows

} and here RT environment is Operating System (OS).

~~If you bundled up some commands~~

* If you take some commands and bundled up into a file and execute this file, this becomes Script.

* When you write Javascript you are basically writing instructions for a Runtime environment.
  → There could be multiple different Runtime environment but the most common or most popular is web-browser.

×——×——×

→ When you have a web browser and making a call to server and server respond with some Html page travels over Http is basically a bunch of text/string.

* Than brawer converts this HTML into Dom (Tree of objects).

* HTML is a static language. If you have just HTML and browser loaded, you get a particular Dom tree and browser builds the tree and renderes the view based on this Dom.

* If you loads the HTML again & again you will get the same Dom & view.

* However HTML comes with javascript. and javascript is a dynamic language.

Once you receive HTML with javascript than browser does the same thing it will create the Dom tree but than executes the Javascript now in Javascript you have the opportunity to use the Dom tree nodes and than modify, you can chang the Dom tree, you can get new nodes added to the Dom, you can removes nodes from the Dom you can do all kind of modifications.

- And than browser renders that modified Dom tree as view.
- So Javascript is a way for us to execute dynamic functionality on browser.

×——×

→ Why learn JS:-

- Client side web development
  - Native Javascript ✓
  - jQuery ✓
  - Angular JS, React, Ember
- Server side development
  - NodeJS ✓
  - Express ✓
- Browser extensions/plugins/Add-ons
- Desktop Applications ✓
- Mobile Applications ✓
- IoT - Applications
- Gaming ✓

→ History of JavaScript:-

- Created by Brendan Eich at Netscape.
- Created to complement java (Java for server-side and JS for client-side)

- JavaScript as it ~~as~~ is out of box is a very forgiving language.

- Standardized as ECMAScript.

× ×

→ Setting up our development environment:-

- We need one editor (like notepad, notepad++, VS code ...) and a browser.

- In browser itself we can write JS in Console and ~~Snact~~ Scratchpad.
  - in console we can execute one by one instruction but in scratchpad we can run bunch of statements together.

× ×

→ Variable Declaration:-

```
var value = 42;
```

OR

```
var value;
value = 42;
```

→ JS doesn't have concept of typed variable.
→ There is no pre-declaration of type required in order to create a variable.

→ Primitive Types:-

- Number
- String
- Boolean
- undefined
- null

One more primitive type added in ECMAScript-6
- Symbol

× ×

→ Number primitive type:-

- Numbers in JavaScript are "double-precision 64 bit format IEEE 754 values".
  (No Integers!)
- So this is all floating point double-precision 64-bit Number.

- Language like Java we have int, float, double we have whole lot of types depending upon precision of the number you want to hold.
- But in JavaScript we don't have that option.
- Every no. we declare even if we declare a number with 1 it is not an integer one, its a double-precision 64-bit value containing the value one.

× ×

→ String:- Sequence of Unicode characters (16-bit)

- (No character type! A character is just a string of length 1).
→ The internal format for string is always UTF-16, it is not tied to the page encoding.
→ String can be enclosed within either single quotes, double quotes and backticks. Backtick allow us to embed any expression into the string.

→ Boolean :- True or false. (only two values)

×——×

No type safety and no strong typing and no type information associated with variables.

```
var a = 10;
var b = 20;
var c = a+b;
var d = "Hello";
var e = true;
console.log(a);
——— b ———;
——— c ———
——— d ———
——— e ———
a = "Hello Javascript";
console.log(a);
```

This behaviour is called as loose typing or weak typing.

×——×——×

→ Undefined type :-

Declaration and Definition :-

↓ asking compiler to create a variable and give it a name `var value;`

↓ assign some value `value = 42;`

→ value is "undefined"

→ once we declare a variable and not provide any definition then Javascript assign a value to it that is "undefined" and type of undefined.

→ undefined type has only value undefined.

×——×

→ null type :- has only value null.

```
var a;
console.log(a); => undefined

a = null;
console.log(a) => null
```

×——×

→ undefined and null Difference : Both are no values.

→ They indicate absense of the value.

=> Types Summary :-

- No need to declare variable type.
- The same variable can be assigned values of different type.
- No scoping information in variable declaration.
- variable and values can be interrogated.

=> The typeof operator:-

In order to identify what the type of variable is we can say typeof and give it value and this operator will give you type of that variable.

Ex var a = 10;
console.log(typeof a); // number

a = "Hello";
console.log(typeof a); // String

a = true;
console.log(typeof a); // boolean

a = null;
console.log(typeof a); // object *
↓ (It shows wrong)
Show

actually it should null because typeof null is null only but it is a bug in Javascript and developers of JS accepted it but now they cannot change it b/c it could cause already projects.

=> Type coerison and === operator:-

→ automatic type conversion by the interpreter is known as coerison.

Ex Concatenation with String values

123 + "4" = "1234"
Number String = String

∞ Execution in Javascript Strait from left.

Javascript automatically converts 123 into String.

→ Ex-1 var a = 10;
var b = 10;
if (a == b)
{
console.log("values are equal");
}

o/p:- values are equal

Ex-2. var a = 10; → Number
var b = "10"; → String
if (a == b)
{
console.log("values are equal");
}

o/p:- values are equal

~~"10" == "10"~~

'==' operation is forgiving it assumes that you by mistake used some other type so it converts a (number) to string and than matches with b and gets true and executes if block.

So this is a overcome by '===' which is introduced by Javascript designer.

Ex. var a = 10;
var b = "10";
if (a === b)
{
console.log("True");
}
else
{
console.log("false");
}

o/p:- false.

checks values & type

So in case of comparison we should use '===' it will be safe with types.

× ×

⇒ * Type coersion Summary:-

Every value in Javascript has a corresponding boolean value.

variable	boolean value.
var a = 10; ✓	true
var a = -10;	true
* var a = 0;	false ✓
var a = "hello";	true
* var a = "";	false ✓
* var a = undefined	false ✓
* var a = null	false ✓
* var a = -0	false ✓

a != 0

var a = NaN ⇒ Boolean(a) → false

- ~~non~~ zero number will be ~~false~~ false, empty string will be false, null, undefined and false will be false.
- Javascript is "flexible" with typing
- [values of all types have an associated boolean value]
- Always use === for precise checks (both value and type).

✓ === → check both type and value

! = → check only value

== → check only value

✓ !== → check both value and type

44 - (And) | >
|| - (OR) | <
! (NOT) | >=
| <=

!:

in JS we have classless object

⇒ Object:- • JS is an OOP language.
• But it's not class based. It's not a class based OOP language. we can create object without classes.

- JS objects are essentially free form. They are not bound to a particular class.
- we can create objects in many ways but easiest way is ~~to create~~ by creating object inline.

```
var myobj = {};  => Empty object
console.log(myobj); // object {}

myobj.prop = "Hello";
Console.log(myobj); // object { prop: "Hello" }

myobj.prop2 = 123;
console.log(myObj); // object { prop: "Hello", prop2: 123 }

console.log(myobj.prop2); // 123
```

→ objects are used to store keyed collections of various data and more complex entities. In primitive types their values contain only a single ~~string~~ thing (be it a string or ~~non~~ number or _)

⇒ we can also use multiword property name, but with " "

```
let user = { "likes birds": true };
```

⇒ Computed properties:-

```
let fruit = prompt("which fruit to buy", "apple");
let bag = { [fruit]: 5, };
alert( bag.~~fruit~~apple); // 5
```

⇒ Object Literal:-

```
var myObj = {
    "prop1" : "Hello",
    "prop2" : 123,
    "prop3" : true
};

console.log(myObj);  // Object { prop1: "Hello", prop2: 123, prop3: true }
console.log( myObj.prop1); // 123

console.log("Accessing property that doesnt exist gives:" +
                                    myObj.prop4);
```

* o/p:- Accessing property that doesnt exist gives undefined.

⇒ Javascript Objects characteristics:-

- Free form - not bound to a class
- Object literal notation to create objects.
- Object properties can be accessed directly.
- New properties can be added on objects directly.
- Objects can have methods.

⇒ The dot and bracket notations:-

- There are two ways to access the property from an object.


```
var myObj = {
    "prop" : "Hello",
    "prop1" : 123,
    "prop2" : true
};

console.log( myObj.prop1);      // 123
console.log( myObj["prop1"]);  // 123
```

⇒ Difference b/w . and [] notation:-

→ '[]' notation is useful when in certain scenario '.' notation doesn't work.

→ Use [] notation when:

(i). property name is a reserved word / invalid identifier

(ii). property name starts with a number.

```
var myObj = {
    "prop" : "Hello",
    "prop1" : 123,
    "1" : "one"
};
```

Cannot access with '.' operator

✗ console.log (myObj.1); | Exception: SyntaxError: missing) after argument list

✓ console.log(myObj[1]) // one

✓ console.log(myObj["1"]); // one

(iii) property name is dynamic

Ex:- var propertyName = "prop1";

✗ console.log(myObj.propertyName); // → will not work

⇒ In this case we need to use [] notation

✓ console.log(myObj[propertyName]); // 123

but in case of '[]' notation it has no idea the property it needs to pull-up on run time only knows. So '.' notation is faster than '[]'

Date 16
Page

(iv) Dot and [] notation can be interchanged

(24) => Nested objects:-

```
var myObj = {
    "prop" : "Hello",
    "prop1" : 123,
    "objProp" : {
        "innerProp" : "Inner property"
    }
};
```

```
console.log(myObj.objProp); // Object { innerProp: "Inner property" }
```

Adding into nested object

```
console.log(myObj.objProp.innerProp); // Inner property
myObj.objProp.newInner = "new inner value";
console.log(myObj.objProp)
```

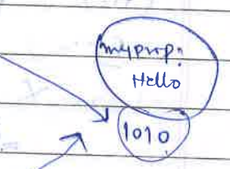
↳ Object { prop: "Hello", prop1: 123, objProp: Object }
↳ innerProp: "Inner property", newInner: "new inner value"

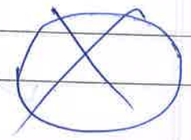
=> Accessing nested object value:-

```
console.log(myObj.objProp.newInner); // new inner value
console.log(myObj.objProp["newInner"]); // new inner value
```

=> Revisiting === for objects:-

```
var myObj = {
    "myProp" : "Hello"
};
var myObj2;
myObj2 = myObj;
```





```
console.log(myObj.myProp); // Hello
if (myObj === myObj2) {
    console.log("variables are equal"); //
}
```

↳ check memory location pointing by obj and obj2

↳ (===) will work same as (==)

=> Revisiting undefined vs null for object:-

```
var person = {
    "fName" : "Koushik",
    "lastName" : "Kothagal"
};
person.age // undefined
```

```
var person = {
    'fName' : 'Shubhan',
    'middleName' : null,
    'lName' : 'Kumar'
};
person.middleName // null
```

=> Deleting properties with the delete operator:-

```
var person = {
    'fName' : 'Shubhan',
    'mName' : null,
    'lastName' : 'Kumar'
};
person.age // undefined
```

```
var person = {
    'fName' : 'Shubhan',
    'mName' : null,
    'lName' : 'Kumar',
    age : undefined
};
person.age // undefined
```

where age property removed from object
if we print object we didn't will not See this age property inside object.

```
delete person1.age;
person.age || undefined
```

=> Introduction Arrays:-

```
var myArrays = [100, 200, 300];
```

myArrays[0]; // 100
myArrays[1]; // 200
myArrays[2]; // 300
myArrays[3]; // Undefined

Adding to array →

```
myArrays[3] = "Java Brains";
console.log(myArrays);
```

[100, 200, 300, "Java Brains"];

=> The Secret behind JavaScript Arrays:-

* • Any JavaScript array Secretly is a JavaScript object.
  • It just has some special properties, but internally it just an object.
  • when we call myArrays.length then actualy we are calling object.property.

myArrays.length // 4
myArrays[4] = 600
myArrays.length // 5

var myArray2 = myArrays;
myArray[3] // "Java Brains".

myArrays // Array [0, 100, 200, 300, "Java Brains", 600]
0: 100
1: 200
2: 300
4: JavaBrains
4: 600
length: 5

accessing like object property

myArrays[0] || 100 => Here JS converting this no → automatically String
Implicit conversion | type coercion.

myArrays["0"] || 100

```
myArrays[400] = "baz";
```

myArrays
0: 100
1: 200
2: 300
3: "Java Brains"
4: 600
400: "baz"
length: 401 → last index + 1

• Arrays are also free form.

myArrays["foo"] = "abc"

myArray
0: 100
1: 200
2: 300
3: "Java Brains"
4: 600
400: "baz"
foo: "abc"
length: 401 → Still 401 So length counts on the basis of last number index + 1

400 + 1 => 401

neglect to count in length

⇒ Wrapper objects :-

Is String is primitive :?

```
var greeting = "Hello world";
greeting.length ⇒ 11
```

typeof greeting // "string" → and it is a primitive type then how can we access .length ?

* The Reason is :- Javascript has equivalent object for each of the primitive data type.

- String primitive has an equivalent String Object.
- When we say greeting.length what Javascript does is that it gets that String and converts it to the equivalent object. When it converted to the object the length property will be available. Than it calls length property of that object not primitive.

- If this (greeting.length) is called after (greeting.length) is still be same strig, because when String object was created it is not assigned to greeting that was temporary object only we used length property and then object was destroyed.

- So the object gets created for a fraction of a second just to make greeting.length work.

- we have four type wrapper object :-

String	Boolean
Number	Symbol

- String has a primitive as well as object
- Number is a primitive as well as Number object
- Boolean — boolean —
- Symbol — symbol —

⇒ Fenctions :- Here sayHello pointing to this function

Ex.1 Function sayHello() { → f^n declarion.
    console.log('Hello');
}
sayHello(); // function call → o/p → Hello

→ Here actually we are creating kind of an object for this function and then giving that object a name called foo.

Ex.2 Function sayHello(name) {
    console.log('Hello' + name);
}
sayHello("Shubham"); // Hello Shubham.

Ex.3 Function sayHello(name, lastname) {
    console.log("Hello" + name + lastname)
}
sayHello("Shubham", "Kumar"); Hello Shubham Kumar

⇒ Flexible argument counts :-

Calling ⇒ sayHello("Shubham"); // Hello Shubham undefined

sayHello("Shubham", "Kumar", 1995); // Hello Shubham Kumar
(1995 ← Ignored)

* • Overloaded function are not possible in javascript.

⇒ Return value of function:-

```
function sayHello(name){
    return 'Hello' + name;
}

var returnValue = sayHello('Shubham');
returnValue // Hello Shubham
```

```
function sayHello(name){
    return;
}

var returnValue = sayHello('Shubham');
returnValue // undefined
```

If we not return anything then also returnVale will be undefined.

⇒ Function Expression:-

- function in JS are first class values
- Functions are values in JS just as much as string is a value, similarly functions is also a value.

```
var a = "Hello";

var f = function foo(){
    console.log("Hello");
};
f(); // Hello; — Fuction foo executed.
f // f foo(){
    console.log("Hello");
}
→ variable executed.

f = 1;
f(); // Exception TypeError f is not a function
```


function

⇒ Anonymous Expressions:-

```
var f = function(){
    console.log("Hello");
};
f(); // "Hello";

f = 1;
f(); // Exception TypeError: f is not a function
```

⇒ Function as argument:-

```
var f = function(){
    console.log("Hello");
};
```

(i)
```
var executor = function(fn){
    console.log(fn);
}
executor(f); // function f(){
                console.log("Hello");
             }
```

(ii)
```
var executor = function(fn){
    fn();
}
executor(f); // Hello
```

(iii)
```
var executor = function(fn){
    console.log(typeof fn);
}
executor(f); // function
```

Ex(iv) var f = function (name.) {
console.log ("Hello " + name);
};

var executor = function (~~fn~~) (fn, name) {
fn(name);
}

executor(f, ~~name~~ "Shubham"); // Hello Shubham

=> Function on objects:-

- Anything you can assign to a variable also you can assign to a object property.

- ~~For name~~ In Javascript objects with properties which could potentially be functions.

Ex. var myObj = {
"testProp" : the
};
myObj.myMethod = function() {
console.log ("Functions in object");
};

myObj.myMethod(); // functions in Object

- Here myMethod is just as an another property.

executes "getfullName" function.

38.

=> Understanding of this keyword :-

var person = {
"firstName" : 'Shubham'
"lastName" : 'Kumar',
"getfullName" : function () {
return person.firstName + " " + person.lastName;
}
};

var fullName = person.getfullName();
console.log(fullName); // Shubham Kumar.

- It looks that its working but there is a risk involved in this. This code is fragile.

This piece of code will be executed after person.getfullName() is executed.

var person2 = person;
person = {};
person2.getfullName(); // "undefined undefined"

- we should not have dependency on person variable, because it may change in our further code, Actually we don't want to get person.fname or person.lastname, we want to get this objects firstName and lastName in which method is declared.

→ For this work we have 'this' keyword who denotes the object where function is declared or denote self or current object.

Date
Page 26

```
var person = {
    'FName' : 'Shubham' -
    'LName' : 'Kumar' -
    'getfullName': function() {
    return this.FirstName + " " + this.lastName;
    }
};

var person2 = Person;
person = {};
Console.log( person2.getfullName()); // Shubham Kumar
```

39' ⇒ Code Exercise :-

40 ⇒ Soln Exceed :-

```
var person = {
    "FName": "Shubham",
    "LName": "Kumar",
    "address" : {
        "Street" : "123 JS street",
        "city" : "JS",
        "State" : "CA"
    },
    "isFromState" : function (state) {
        if (this.address.State === state) {
            return true;
        }
        else
        {
            return false;
        }
    }
};

Console.log( person.isFromState("CA")); // true
```

return (this.address.State === State);

Date
Page 27

it is arguments → an object not an array → this is implicit or hidden argument. It is equal like in Java Var-arg or params in C#
→ this is also available for free as arguments.

41' ⇒ Default function Arguments :-

```
var add = function (a,b) {
    console.log (arguments);
    return a+b;
};

console.log (add( 10,30)); // Arguments: {
                                 0: 10
                                 1: 30
                                 callee: add()
                                 length: 2
                                 proto: object }
40

Console.log(add(10,30,2,3,4,5));
    ⇒ output → Arguments{ 0: 10, 1: 30, 2: 2 ... more}
40
```

(now not required if we are using argmts.)

```
var add = function (a,b) {
    var i, sum = 0;
    For(i=0; i<arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
};

console.log (add(10,30,2,3,4,5)); // 54
```

Note :- The arguments value is not an array. although its looks like an array.
- It's actually an object. It's basically using [] operator to look up the values.
- If you try some of the other operations that would work on array it wouldn't work on arguments value.

arguments → is not an array it is an object