JAVASCRIPT (It was ECMA script all along)

→ Browser language to add interactivity / dynamic effect to webpage.

It is a single-threaded, cross-platform, object-oriented, concurrent programming language.

⇒ Data Types in JS:

I. Numbers: Floating point numbers, for decimal & integers
II. String: Sequence of chars, used in text.
III. Boolean: True / False
IV. Undefined: Data value of variable that does not have a value yet.
V. Null: Non-existent. (It is an object still).

JS is dynamic typing → Automatically asign type to variables.

Note: In JS, every number is floating-point.
Note: Variable names can start with '$', '_', any letter.

⇒ TYPE COERCION:

JS converts type from one to another automatically.

e.g. consde.log (Var1 + " " + Var2)
↓ Boolean

Variable Mutation:

```
var age = 4;
age = "Hey";
```

null == undefined, null !== undefined.

⇒ Falsy & Truthy Values & equality operators

Falsy → Undefined, null, 0, '' (empty string), NaN

Truthy: ! Falsy.

Note: == does type coercion

23 == '23' true

=== doesn't

23 === '23' false ✓

⇒ Functions:

I. Function declaration

function whattodo (~~arguments~~ parameters) { }

II. Function expression.

var whattoDo = function(parameters) { };

✱ JS expressions are something that always returns a valu

✱ JS statements are something that do some action.

✱ JS variables are immutable

✱ JS objects are mutable.

⇒ Arrays:

```
var names = ["a", "b", 23, true];
var years = new Array(10, "a", true);

names.length
names[1]
names[10]   (Have 6 empty in between)

names.push(x)        (Add At end)
names.unshift(y)     (Add At front/start)

names.pop()          (Remove from end)
names.shift()        (Remove from start)

names.indexOf(x)
    returns first element that matches else return -1
```

⇒ Objects & Properties:

object

```
var john = {
    firstName: "SK"
    lastName: "Ag"
    DOB: 2000
    isMarried: false
}
```

properties — key-value pair.

```
john.<property-key> = <value>
john['<property-key>']

var john = new Object();
```

Object Methods:

```
var John = {
    fName:
    Age:
    calcAge: function( ){ return this.age. }
}
John.calcAge(Arguement);
```

⇒ JAVASCRIPT AND DOM:

Javascript ⟷ DOM

Interaction/ Manipulation

Basic JS

→ Background:
called JS, but due to copyright issues, name is ECMA Script

→ Statement vs expressions:
↓ Do things
↓ produce values.
eg. var foo 3+7

→ Semicolons:
Optional in JS, but JS can guess wrong about end of statement

→ Identifiers & Variable Names:
Names that play various syntax roles in JS.
A variable name is an identifier
(Can start with any Unicode letter)
'$', '_'

→ Values:
All values in JS has some properties.

value.propKey.
eg. str.length

Values
- Primitive
  - → booleans
  - → numbers
  - → strings
  - → null
  - → undefined
- Objects
  - everything else.

These are compared
var a=2
var b=2
a===b
true

They are unique
var obj1 = {}
var obj2 = {}
obj1 === obj2
false

DATE

→ Primitive Values:

⇒ ✱ Content is Compared by value

4 === 4 ✓

4 === 5 ✗

✱ Always immutable

Properties can't be added, removed or changed

eg- str.length = 1

No change in str length

⇒ Objects:

I. Plain Objects:

```
{
    name : "sk"
}
```

Property — Value

II. Arrays:

['a', 'b', 'c']

III. Regular expressions

/^a+b+$/

✱ Content is compared by reference

{} === {} ✗

var obj1 = {}

var obj2 = {}

obj1 === obj2 ✗

obj2 = obj1

obj1 === obj2 ✓

✱ Always Mutable.

⇒ undefined & Null.] They have no properties ☹

↓ No value. ↓ No object

var foo;
↑ undefined.

if (x == undefined || x === null) {
}

if (!x) {
}

✱ both undefined & Null are considered falsy.

⇒ Categorizing Values using typeof and instanceof

typeof value

{} → object
true → boolean
undefined → 'undefined'
null → 'object'

It is a bug, as null is a primitive value.

value instanceof constructor (Doesn't work for primitives use wrapper objects)

var b = new Bar()
b instanceof Bar ✓
{} instance of Object ✓
[] instance of Array ✓
Object

null instance of Object
False

→ Booleans: Operators producing booleans are:

Binary logical → && , || (Returns first operand that is true)

Prefix logical → !

Comparison

Equality → ===, !==, ==, !=

Ordering → >, >=, <=, <

Values Interpreted as False	Values Interpreted as True
→ Undefined, Null	everything else
→ false	e.g. {}, Infinity
→ 0, NaN	[]
→ ''	Boolean({}) → true

NaN: A number generated by arithmetic operator on undefined

'' : ↓ empty String

→ Numbers: All Number in JS are floating-point

1 === 1.0 ✓

✱ NaN is number

✱ → Infinity " "

pow(2, 1024) ↖ 3/0 ✓

✱ -Infinity " "

→ Operators

+ Addition, -, /, %, *, ++, --

-n (Negate value)

✱ +n (Converts to Number)

e.g. +'123' = Number 123

→ Strings :

Can be declared via string literals
single / Double Quotes.

'+' operator concanates the string.

+= can be used

String methods.

'abc'

str.slice(1) → 'bc'

str.slice(1, 2) → 'b'

str.indexOf('bc') → 1

str.toUpperCase() → 'ABC'

str.trim()

→ Statements :

I. Conditionals :

```
if ( ) {
}
else {
}
```

if
else if
else

* No need to use {} if only a single statement
* {} creates a block of zero or more statements

```
switch (x) {
    case 1:
    case 2:
    default:
}
```

II. Loops:

for, while, do while

→ Functions:

I. Function Declaration

```
function add (x,y) {
    return x+y;
}
```

II. Function Expression

```
var add = function (x,y) {
    return x+y;
};
```

It is a mechanism in which Funct or variables (declarations) are moved to the top of their scope before code execution.

✗ Function Declarations are hoisted, but func^n expression isn't.

eg:

```
function bar() {
    foo()
    function foo() {
    }
}
```

Works

```
function bar() {
    foo()
    var foo = function() {
    };
}
```

Not

✱ Variable Arguments

```
function f() {
}
var args = f('a', 'b', 'c')

args[0]  // 'a'
args[1]  // 'b'
```

Can be accessed via "arguements" array

Not an actual array but array-like.

In case of Too Many :
Excess are ignored
but "arguements" array knows them all

In Case of Too Few :
Undefined is given to them
"Arguements" array ~~doesn't know~~ undefined.

→ Exception Handling :

```
function err() {
    throw new Error("I am an error");
}

try {
    err();
} catch (e) {
} finally {
}
```

Not required but always execute whatever the result.

→ Strict Mode :
Enables more warnings & makes JS a cleaner language

```
'use strict';
```

→ Variable Scoping & Closures:

* I. Variables are function-scoped, not block-scoped.

(e.g.

```
function foo() {
    var x = 5;
    if (x === 0) {
        var y = 2;
    }
    console.log(y)
}
```

// No error, undefined if x === 0, 2, if x = 5

* Variables (declarations) are hoisted.

Their declaration is moved to beginning but assignments that it makes stays put,

eg

```
function fool() {
    console.log(n);
    if (false) {
        var n = 3;
    }
}
```

hoisted ⇒

```
funcn foo() {
    var n;
    console.log(n);
    if (false) {
        n = 3;
    }
}
```

~~* (Closures) (Later)~~

* Introducing IIFEs (immediately invoked func^n exp.)

Since variables are function scoped, a scope can be

created for variable that makes it behave like blocks.

```
if (true) {
    (function() {      // open IIFE
        var x=3;       // inside
    }());              // close IIFE
}
```

↑ required semicolon.

* Closures (Functions stay connected to their Birth Scopes):

```
function createInc(startValue) {
    return function(step) {
        startValue += step;
        return startValue;
    };
}

var inc = createInc(5)

inc(1)
⇒ 6
inc(2)
⇒ 8
```

start value is changing.

→ Objects & Constructors:

```
var obj = {
    name : "Jane",        // name: property (stored as string), "Jane": value
    describe: function() {
        return this.name;
    }
};
```

Function-valued property also called as method.

```
obj.name          // Jane
obj.describe      // shows value of func^n
obj.describe()    // Jane

obj.name = "Hey"  ✓
obj.unknown;      // undefined
```

* name in obj // true
  (in operator)
* delete obj.name. // removes property name
* obj["name"], square box notations are used for string
  They allow computing the key of a property.

e.g. var x = "name";
obj[x] ✓
obj["n" + "ame"] ✓

⇒ Extracting Methods:

var func = obj.describe

func() // May works but this becomes an error.

So, use

var func = obj.describe.bind(obj)

func() // same value of this

bind() creates a new function whose 'this' always has the given value

* Every function has its own variable 'this'

⇒ Constructors:

Later.

→ Arrays:

Sequences of elements that can be accessed via integers starting from zero.

```
var arr = [1, 2, 'a']

arr[0] = 5

arr.length
```

2 (in) arr

↓ Index (does index 2 ~~exist~~ in array)

```
arr.foo = 5
arr.fool = function () {
    console.log ("Hey");
}

"foo" in arr // true
```

This makes it array of elements + properties + Methods.

Note: All methods defined below works on only elements

```
arr.slice (1)
arr.slice (1, 2)
arr.push ('x')
arr.pop ()      // remove last
arr.shift ()    // remove first
arr.unshift (3) // prepends an element
arr.indexOf ('a')   // 2
"     "     (10)    // -1
```

⇒ Iterating over Arrays:

May be ignored

```
arr.forEach ( function (elem, index) {
    } )
```

PAGE
DATE

Advanced JS

Ch-7

→ Statements & expressions:

⇒ Statements Vs expression

```
if (male)
    age = 10
else
    age = 11
```

} statement

or

```
var age = (male ? 10 : 11);
```

} expression.

⇒ expression that looks like statement:

i) Object literals (expressions) looks like blocks (statements)

```
{
    foo: bar(3,5)
}
```

ii) Named function expression looks like function declaration.

```
function foo() { }
```

So, in order to prevent ambiguity, JS doesn't let you use object literals & func" expressions as statements.

→ Semicolons:

i) Statements are terminated by semicolons

ii) Except statements ending with blocks.

PAGE
DATE

e.g.

```
while (a>0) {
    a--;
}
```

```
do {
    a--;
} while (a>0);
```

```
function foo() {
}
```

```
var foo = function() {
};
```

→ Strict Mode :

'use strict';

* Variables assigned without declaring creates a global variable strict mode prevent that.

* Functions must be declared at the top level of a scope.

* Usually this is window in browser but in strict mode, it is undefined.

* In sloppy mode, str.length=7 or any read only property can't be changed but can be written.
In strict, it gives error.

* Octal literals are not allowed in strict mode, e.g. 010

DATE

Ch-8

→ JavaScript's Type System:

static → 'At Compile Time'
dynamic → 'At runtime'

i) JS is dynamically typed, types of variables are generally not known at compile time.

ii) JS has a very limited kind of dynamic type checking i.e. give some kind of error or exception if a value used in an operation is incorrect.

e.g. var foo = null
foo.prop] error

var bar = {}
bar.prop] No error :gives undefined instead

iii) Coercion: Implicit type conversion
e.g. "3" * "4"
= 12

For Null → It is coerced to 0 with Number
e.g. Number(null) = 0
5 + Null = 5

For undefined → It is coerced to NaN
Number(undefined) = NaN
5 + undefined = NaN

→ Wrapper objects for Primitives: Boolean, Number, String

e.g.

* type of new String ('abc')
  'object'

  new String ('abc') === 'abc' ✗

  new String ('abc') == 'abc' ✓

  String (123)
  = '123'

* 'abc' instance of String // Never true for primitives
  false

Wrap	Unwrap
new Boolean (true)	new Boolean (true). valueOf ()
new Number (123)	↓
new String ('abc')	true

* JS own ToPrimitive() Method:

Step 1: If input is primitive, return it.

Step 2: If input is an object. Call input.valueOf().
If result is primitive return it.

Step 3: Call input.toString(), if result is primitive, return it

Step 4: Throw a TypeError.

Note: If preferred Type is String, step 2 & 3 are swapped.

PAGE
DATE

Operators

Ch-9

All operators coerce their operands to appropriate type.

e.g.

[1,2] + [5,6] = "1,25,6

can be converted only to String gives NaN with Number

I. Assignment Operators:

```
var x = value;
obj. propkey = value;
obj ['propkey'] = value;
arr [index] = value;
```

II. Compound Assignment:

+=, -=, /=, %=, *=
>>=, <<=, &=, ^=, |=
>>>= check

III. Equality operators:

=== strict Equality

!== Negation of strict equality
Strict Inequality

Fails for NaN === NaN
else good

== !=
Normal
1. Same type, strict
2. Undefined & null, then they are equal
3. string & a number, converts string to number then strict

4. A boolean & Non-boolean
Converts boolean to Number (1 or 0)
then strict

5. Object & (String or Number)
Converts to Primitive, then strict.

6. else all false

eg. "" == false ✓

→ 2 == true (1) 2 === 1 X

→ 2 == false (0) 2 === 0 X

→ 1 == true 1 === 1 ✓

→ '' == false (0 === 0) ✓

→ 'abc' == true → 'abc' == 1

↓

NaN === 1 X

→ '123' == 123 → 123 === 123 ✓

→ '\n\t 123\r' == 123 ✓ (This is a pitfall)

→ {} == '[object Object]' ✓ (" ")

→ [] == 0 (" ")

Ordering operator: (i) Convert both to primitives.

<, <=, >, >= (ii) If both are string, compare lexicographically.

(iii) Convert both to numbers & compare them.

(+) Plus operator:

(+) check for +[] === 0 (unary operator converts to Number)

(i) Convert both operands to primitives

(ii) If either is string, convert both to string & concate them.

(iii) Convert both to numbers & return sum

VI. Special operators:

(?:), (,), (void)

↓

does what semicolon does for statements (evaluates both and returns result of right)

<condition>? <if_true>: <if_false>

<left>, <right>

void <expr.>

↑

evaluates expression & returns undefined.

void 4 + 7

= Undefined + 7 = NaN

void (x=5)

= undefined // x=5

VII. Operators on Numbers, Booleans & Objects:

This is explained later.

Booleans

Ch-10

Boolean (value) → converts to Boolean
Boolean (new Boolean(false)) → returns true, as objects are true.

I. Logical operators

&& (AND), Or (||), and Not (!)

Binary logical operators are

a) Value-Preserving:
They always return either one of operands, unchanged

e.g. 5 || 3 → returns 5 , 6 && 3 → returns 3

b) Short-circuiting:
The second operand is not evaluated if first operand determines the result.

Numbers

Ch-11

IS treates all numbers as floating point numbers (double-64 bit)

* A number literal can be an integer, floating point, or hexadecimal

35
3.141
0XFF // 255

eX
5e2 = 5×10^2
1e3 = 10^3

DATE

→ Invoking Methods on literals

123..toString() 123 _ .toString()
(space)

123.0.tS() (123).tS()

→ Converting to Number

Number(value) / +value

undefined → NaN
null → 0
A number → Number
boolean → 0 → false
1 → true
string → Parses no. in string ('' → 0)
Object → ToPrimitive()

→ parseFloat(str)

i) Converts str to string
ii) Trims leading whitespaces
iii) Parses longest prefix that is a floating-point number.

e.g. parseFloat(true) → parseFloat('true')
NaN
parseFloat('') → NaN
So, Number(value) is better.

→ Special Number values:

i) NaN

The error value NaN is a number value

→ A number can't be parsed
→ An operation failed
→ One of operands is NaN

✗ NaN === NaN is false

✗ isNaN (x) and typeof x === 'number' checks if it's a number

ii) Infinity

The error value Infinity is caused by

→ Too large magnitude
for (2, 1024)
→ Division by zero

Note: Infinity − Infinity = NaN
Infinity / Infinity = NaN
Infinity + Infinity = Infinity
Infinity * Infinity = Infinity

✗ isFinite (x)

iii) Two zeros

Just consider them one.

⇒ Internal Representation of Numbers:

Sign	+	Exponent	+	Fraction
1 bit		11 bits		52 bits
(63 bit)		(62-52)		(51-0)

Value of Number = $(-1)^{sign} \times$ 1. fraction $\times 2^{exp-1023}$

(1. fraction → number is in binary)

Special Exponents:

$-1023 < exp < 1024$

↓ Reserved for 0 ↓ Reserved for NaN & Infinity

* Now, due to numbers being stored as floating-point, comparing non-integers may lead to imprecision. Use this function.

```
var EPSILON= Math.pow(2, -53)
function epsEqu(x,y) {
    return Math.abs(x-y) < EPSILON
}
```

⇒ Convert 0.1 to 64 bit floating point representation.

0.1 * 2 = 0.2 — 0.0
0.2 X 2 = 0.4 — 0.00
0.4 X 2 = 0.8 — 0.000
0.8 X 2 = 0.6 — 0.0001
0.6 X 2 = 0.2 — 0.00011

$= 0.0\overline{0011}$

So $(0.1)_{10} = (0.0\overline{0011})_2$

$= (1.\overline{1001})_2 \times 2^{-4}$

Mantissa = 10011001100110011001 (52 bits)

Exponent = $2^{K-1} - 1 + exp$

$= 2^{11-1} - 1 + (-4)$

$= 1019$

$= (01111111011)_2$

* For 0 it's impossible to normalize, so JS stores

0 → Fraction

−1023 → Exponent

* Math.pow(2, 1023) ✓
Math.pow(2, 1024) → Infinity

* Maximum value of double = $1.\underbrace{11111\ldots}_{52\text{ bits}} \times 2^{1023}$

$= 1.8 \times 10^{308}$

⇒ Safe Integer : (is SafeInteger(a))

Imagine above DBL_MAX value with have many precision errors. This happens for integers too.

So,

$(-2^{53}, 2^{53})$

Safe range.

Note: Bitwise operations are 32 bit operations only.

Eg. $1 << 31 = 2^{31}$

$1 << 32 = 1$

$1 << 34 = 4$

⇒ Converting to Integers

Method 1: Math.ceil(), Math.floor(), Math.round()
= Math.ceil(x + 0.5)

Method 2: Inbuilt ToInteger(). (Not available)

Method 3: 32-bit Integers (only) using Bitwise

Recommending order

⇒ x | 0 returns x as 32 bit
⇒ x << 0 Signed 32-bit
⇒ x >>> 0 unsigned 32-bit
⇒ x >> 0 Signed 32 bit

Method 4: parseInt(str, radix) (default 10)
radix → optional, base
Integer → String → Integer
So, costly performance
except when string starts with 0x → base 16

⇒ Operators

1. Arithmetic +, -, *, /, %, ++, --
+ → plus operator (refer back)
% → remainder operator

Remainder operator = Sign of first operand
Modulo operator = Sign of second operand
(Not supported in JS, C++)

Eg. -5 % 3
= -2
(Normal case)

but in JS -5 % 3
= -2
So use Math.abs

II. Bitwise &, |, ^, &&, ||, ☺, <<, >>, >>>

☺ → doesn't exist

>>> → Unsigned right shift

⇒ Function Number:

(i) Number ('123')
type of Number('123') // Number

(ii) new Number ('123') // object

→ Number constructor properties.

Number.MAX_VALUE = 1.79769e+308
Number.MIN_VALUE = 5e-324
.NaN
.NEGATIVE_INFINITY.

Ch-12

STRINGS

I. Escape characters:

'\', '+' for escaping

\b → backspace
\f → form feed
\v → vertical tab
\n → new line
\t → horizontal tab
\r → carriage return

II. Character Access:

s.charAt(ind) s[ind]

→ Converting to String:

undefined → 'undefined'
null → 'null'
boolean → 'true', 'false'
Number → '3.141'
object → call ToPrimitive (value, String)

→ Manually converting to string

"" + value
value.toString() (Doesn't work for undefined & null)
String (value)

→ Concatenating strings:

(i) The Plus operator (Details on previous page)
(ii) arr.join (seperator)
↑ converts whole array to string

→ String Prototype Methods:

Extract

string.charAt (ind)
string.charCodeAt (ind)
string.slice (start, end) ← optional
string.substring (start, end)
string.split (seperator, limit)

Can Handle negative values

Transform
- string.trim()
- string.concat(x, y, z)
- string.toLowerCase()
- string.toUpperCase()

Search & compare
- string.indexOf(searchString, startIndex) (default is 0)
  - startIndex: optional
- string.lastIndexOf(searchString, pos)
  - pos: default is end

Ch-13 STATEMENTS

(or while, for in, if else, do while)

for object

Ch-14 EXCEPTION HANDLING

```
try {
    throw new Error("Some error")
}
catch / finally {
}
```

(or both)

Note: Finally always execute except browser window/ code stops try is not reached

→ Error Constructors:

I. Error (Generic constructor)
All others are sub constructors.

II. RangeError → Indicates a numeric value has exceeded the allowable range.
e.g. new Array (-1)
(Invalid array length)

III. Reference Error: indicates invalid reference value has been detected
e.g. Unknown Variable
(not defined)

IV Syntax Error, Parsing error

V. Type Error → expected type is different than actual type of an operand.
e.g. undefined.foo
(can't read property foo of undefined)

Properties of Error
→ message
→ name
→ stack (Bottom ↓ Top)

Ch-15

FUNCTIONS

→ Three Roles of Functions in JS:

I. Non method function ("normal function")
II. Constructor (new Date())
III. Method (obj.method())

→ Params vs Arg.

function foo(param1, param2){} | foo(3, 7)
args

→ Defining Functions: (All. func^n^s are obj instances of Function)

func^n^ instanceof Function
function instanceof Object

i) Func^n^ Decl. } Explained previously
ii) Func^n^ exp }
iii) Function Constructor

Evaluates its code stored in strings (😬)

new Function ('x', 'y', 'return x+y');

Ch-16

Variable Scoping & Closures

Check for "environment"

Ch-7

Objects And Inheritance

Layer 1: Single objects
Layer 2: The prototype relationship b/w objects.
Layer 3: Constructors - Factories for Instances.
Layer 4: Inheritance b/w constructors.

* Single Objects:

Refer previous intro

⇒ Converting Any Value to an Object

Object() {} or new Object()

undefined → {}
null → {}
A boolean bool → new Boolean(bool)
a number num → new Number(num)
a string str → new String(str)
object → obj

⇒ this as an Implicit Parameter of Functions & Methods:

Note: Normal func's don't require "this" i.e. they have no use cases but still,

Sloppy mode

```
function x() {
    return this;
}
```

Returns Global Object

PAGE

PAGE
DATE

Strict Mode

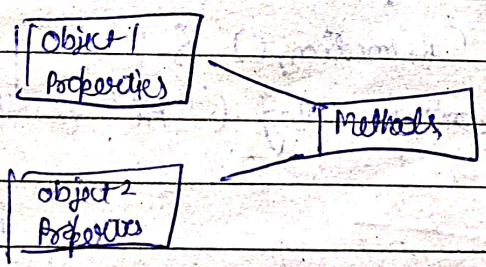
```
function x() {
    'use strict';
    return this;
}
undefined
```

Methods

```
var obj = {
    method : function() {
        'use strict';
        return this;
    }
};
return obj;
```

⇒ Controlling functions more with call(), apply() & bind().

function is an object so it has above mentioned methods.



I. call :

```
obj.sayHello.call(" ");
```

func-name.call(obj, [args])

- obj → this value
- [args] → Not actual array

```
obj.sayHello.call(obj, args);
```

II. apply:

func.apply(obj, [args]);

↓

Actual array.

III. bind:

Performs partial func application

var bound = func.bind(obj);

bound();

Partial call func

var bound2 = func.bind(obj, a, b);

bound2(c,d); // Called with a, b, c, d

var obj = {

firstName: "Colt",

sayHi: function() {

setTimeout(function() {

console.log("Hi" + this.firstName);

}, 1000);

}

};

↑

Now window

'this' lose context, so we need to bind it.

obj.sayHi.call(obj);

obj.sayHi.apply(obj);

var bnd = obj.sayHi.bind(obj); // or //

bnd();

sayHi : function() {

setTimeout (function() {

console.log ("Hi" + this.firstName);

}.bind (this), 1000);

⇒ OOP

A programming model based around the idea of objects.

JS doesn't have "classes" built into it, so we use func" & objects.

I. new Keyword:

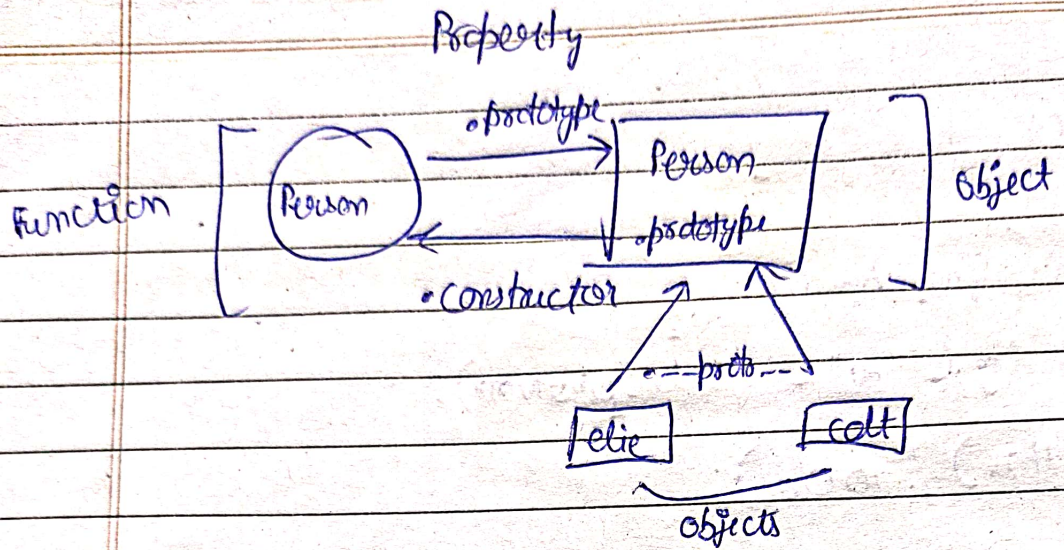
var obj = new House (arg1, arg2);

function House (arg1, arg2) {
this.arg1 = arg1;
this.arg2 = arg2;
}

'New' keyword does:

→ Creates empty objects
→ Sets 'this' keyword to be that empty objects
→ Adds line 'return this' to end of func".
→ It adds a property onto empty object called "__proto__"

II. Prototypes:



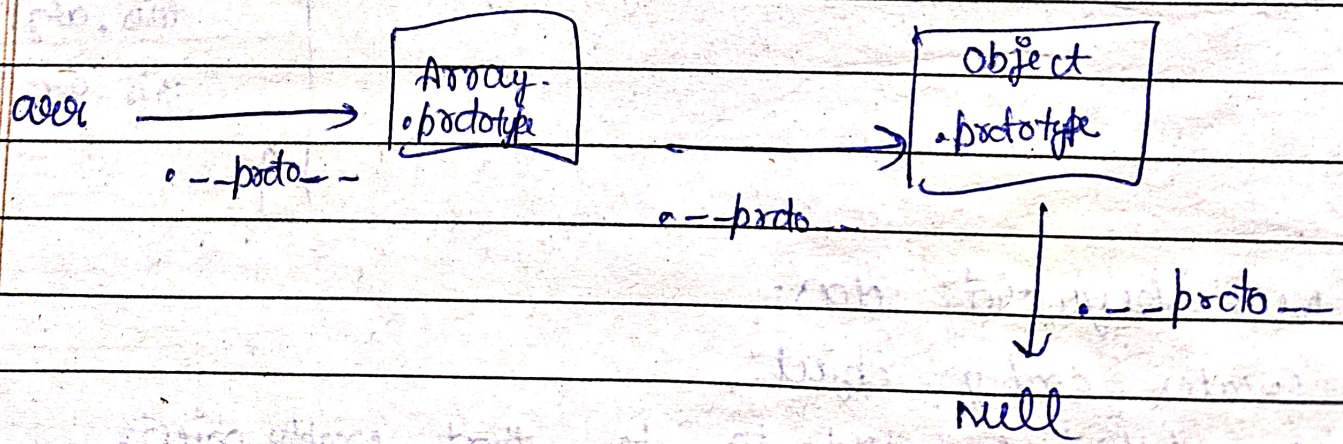
elie.__proto__ === Person.prototype

Person.prototype.constructor === Person.

e.g.

Notes: Prototype is shared among all objects created by that constructor function.

e.g. arr.__proto__ === Array.prototype.



Prototype Chain

REACT (ES6 only)

Javascript Library for building user interfaces.

⇒ Fundamentals of building Single page Applications:

I. Async Foundation:

Callback Function → A func^n that is passed into another func^n as a parameter, then invoked by ther func^n.

Higher order Func^n → A func^n that accepts callback func^n as a parameter.

Callbacks are used for
→ Advanced Array Methods
→ Browser Events
→ AJAX
→ React.

e.g. forEach func^n

Func^n Definition

```
function forEach(array, callback){
    for(var i=0; i<arr.length; i++){
        callback(arr[i]);
    }
}
```

Func^n Call

```
forEach(arr, function(number){
    console.log(number*2);
});

function callback(currEle, currIdx, array)
```

⇒ Stack
(Keeps track of func^n invocations),

⇒ Heap
(An area in memory where your data is stored.

⇒ setTimeout
A func^n that asynch. invokes a callback after a delay in ms.

Cancel setTimeout. → clearTimeout] timerId

⇒ setInterval
A func^n that continually invokes a callback every X ms,

Cancel → clearInterval] intervalId

⇒ Promise :
A promise is an object that represents a task that will be completed in future.

```
var p1 = new Promise ( function (resolve, reject) {
        resolve ([1,2,3,4]); reject ("error"); //Asynch Code
});
p1.then ( function (arr) {
        consle.log (arr);
}).catch ( function (data) {
        consle.log (data);
});
```

⇒ Promise Chaining:

To avoid callback hell in async. code.
↳ Hard to read

Return Promise

```
p1.then ( function(data) {
    return new Promise ( function (resolve, reject) {
        // Async resolve. or reject.
    });
}).then ( function (data) {

});
```

Return data

```
p1.then ( function (data) {
    return data*2;
}).then ( function (data) {
    console.log (data);
});
```

In practice, you will be given promises that are returned you.

* JSON.parse (____)
  JSON data to javascript objects.

Promise.all

```
Promise.all ([p1, p2, ...])
  .then ( function (movieArr) {
  });
```

II. Fetch:

```
fetch (url, {
    method: 'POST',
    body: JSON.stringify ({
        name : 'blue'
        login : 'blue cat',
    }),
    headers :
})
.then ( function (res) {
})
.catch (function (err) {
})
```

only runs when there is problem with request like URL doesn't exists Internet.

III. Ajax:

```
$.ajax ({
    method
    url
    data
    headers
}).done
.fail
.always
```

IV. Axios:

```
axios.get (url)
.then
.catch
```

II. Advanced JS funcⁿs :

(A) Advanced Array Methods :

→ forEach

Iterates through an array

Runs a callback funcⁿ on each value in array

returns undefined.

```
arr.forEach(function(value, index, array) {

});
```

→ map

As forEach returns undefined, we may need to transform array, so we use map.

Creates a new array

Iterates through an array

Runs a callback funcⁿ for each value in array

Adds the result of that callback funcⁿ to new array

Return new Array

```
arr.map(function(val, ind, arr) {
    return value * 2;
});

// [2,4,6];

function map(arr, callback) {
    var newArr = [];
    for(var i=0; i< arr.length; i++) {
        newArr.push(callback(arr[i], i, arr));
    }
    return newArr;
}
```

→ filter:

Creates new array
Iterates through an array
Runs a callback func" on each value in the array
If callback returns true, element added to newArray
else Not

Return new Array.

```
arr.filter(function(val, ind, arr) {
    return true/false;
});

function filter(arr, callback) {
    var newArr = [];
    for (var i=0; i<arr.length; i++) {
        if (callback(arr[i], i, arr))
            newArr.push(arr[i]);
    }
    return newArr;
}
```

→ some:

Iterates through an array
Runs a callback on each value in the array
If callback return true for atleast one single value, ret true
else return false

```
arr.some(function(val, ind, arr) {
    return true/false;
})
```

```
function some (arr, callback) {
    for (var i=0; i<arr.length; i++) {
        if (callback (arr[i], i, arr)
            return true;
    }
    return false;
}
```

→ every :
opp. of some

→ reduce:

- Accepts a callback func^n & an optional second parameter
- Iterates through an array
- Runs a callback on each value in the array.
- The first parameter to the cb is either first value in array or optional second parameter.
- The first parameter to the cb is called accumulator
- The returned value from callback func^n becomes new value of accumulator.

arr.reduce (function (accumulator, nextValue, index, arr
whatever returned inside here, will be the value of
acc. in next iteration.

}, optional_second_parameter);

e.g. arr = [1, 2, 3, 4, 5]

```
arr.reduce ( function (acc., nextVal) {
        return acc. + nextVal;
});
```

acc.	next Val	returned Value
1	2	3
3	3	6
6	4	10
10	5	15

returns 15

(B) Closures :

A closure is a func^n that makes use of variables defined in outer func^n that have previously returned.

```
function outer (a) {
        return function inner (b) {
                return a+b;
        }
}
outer (5)(5); // 10
var storeOuter = outer (5);
storeOuter (10); // 15
```

* Closures can help for private variables.

e.g.

```
function counter() {
    var count = 0;
    return function inner() {
        count++;
        return count;
    }
}

var counter1 = counter();
counter1();   // 1
counter1();   // 2

var counter2 = counter();
counter2();   // 1
counter2();   // 2
```

more privacy

```
function classRoom() {
    var instructors = ["A", "B"];
    return {
        getInstructors: function() {
            return instructors.slice();
        },
        addInstructor: function(instructor) {
            instructors.push(instructor);
            return instructors.slice();
        }
    }
}
```

(C) ES 2015 : (ES6)

(i) const

Can't redeclare also can't reassign
& Mutate if it is an object.

e.g. const arr = [1,2,3,4]
arr.push(5);

(ii) let (Block Scope)

Can't redeclare, but can reassign

```
if (variable === "Hey") {
    let number = 1;
}
newVar; // Not defined
```

(iii) Template Strings :

` ${} `

Also can do multi line strings.

(iv) Arrow Functions;

```
var add = (a,b) => {
    return a+b;
}
```

One liner

```
var add = (a,b) => a+b;
```

* Arrow func". don't get their own 'this' keyword.
* They get the value this just outside its context.
* Arrow func" don't get arguments array.

e.g.

```
var instructor= {
    firstName: "Elie",
    sayHi : function() {
        setTimeout ( function () {
            consde.log ('Hello ${this.firstName}');
        }, 1000);
    }.bind (this)
}
instructor.sayHi() // Hello undefined
                   (use bind keyword)
```

but with arrows,

```
var instructor = {
    firstName : "Elie",
    sayHi : function() {
        setTimeout ( () => {
            consde.log ( this.firstName);
        }, 1000);
    }
}
// Elie
```

get's context from here

(v) Default Parameters

```
function add (a= 10, b=20) {
    return a+b;
}
```

(vi) for ... of

```
var arr = [1,2,3,4,5];
for (let val of arr) {
    console.log(val);
}
```

* Can't be used on objects b/c it don't have Symbol.iterator method implemented
* Can't access an index.

(vii) Rest operators

```
function printArg (a, b, ...c) {
    console.log(a);    // 1
    console.log(b);    // 2
    console.log(c);    // [3,4,5]
}
printArg(1,2,3,4,5)
```

(viii) spread operator:

```
var arr = [...arr1, ...arr2]
```

spreads array.

9. Object Shorthand Notation

```
var instructor = {
        firstName,
        lastName
}

var instructor = {
    sayHi() {
        return "Hello";
    }
}
```

10. Destructuring
extracting values from data stored in objects & arrays.

```
var { firstName, lastName } = instructor;

var { firstName : first , lastName : last } = instructor.
```

Array Destructuring

```
var [a, b, c] = arr;
```

eg. swapping values

```
[a, b] = [b, a]
```

11. OOP in JavaScript

i) class

New Reserved keyword
Abstraction of constructor func & prototypes
Creates a constant - can't be redeclared
Doesn't hoist
still use 'new' keyword.

```
class Student {
    constructor (firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

var elie = new Student ('Elie', 'Schoppik');
```

Instance methods

```
class Student {
    sayHello() {
        return "Hello";
    }
}
```

Mimic of Student.prototype.sayHello

Class methods

```
class Student {
    static isGood (obj) {
        return obj.isGood;
    }
}
```

Mimic of Student.isGood

ii) Inheritance

```
class Person {

}
class Student extends Person {

}
```

Mimic of

```
Student.prototype = Object.create(
    Person.
    prototype);
```

iii) Super

```
class Student extends Person {
    constructor (firstName, lastName) {
        super (firstName, lastName)

    }
}
```

12. Maps (Hash map of C++)

Keys can be of any data type, unlike objects.

```
var firstMap = new Map;
firstMap.set (key, value);
firstMap.delete (key)
firstMap.size ();
firstMap.get (key);
firstMap.forEach (v => console.log(v));
firstMap.values ();
firstMap.keys ();
```

/ can be iterated

13. Sets

All values in a set are unique.

Any type of value can exist in a set

Created using new keyword.

```
var s = new Set ( [1, 2, true, 4] )

s.add (key)

s.delete (key)

s.has (key)

s.size
```

Can be iterated

14. Generators

→ A special kind of func^{n} which can pause execution & resume at any time.

→ Created using a *

→ When invoked, a generator object is returned to us with the keys of value & done.

```
function* pauseAndReturn (num) {
    for (let i=0; i<num; i++) {
        yield i;
    }
}

var gen = pauseAndReturn (10);

gen.next()    // { value : 0, done: false }

gen.next()    // { value: 1, done: false }

gen.next()    // { value: undefined, done: true }
```

(D) ES 2016 & ES 2017:

1. Exponentiation operator:

Math.pow (2,4)

2 ** 4

a **= 4 // a = a ** 4

2. Array Includes:

arr.includes (key)

str.includes (substr)

3. padStart & padEnd:

str. padStart (arg1, '!');

arg1 → total length of new string

'!' → what to pad from start (default = '' whitespace)

str. padEnd (arg2, '!');

4. Async Functions:

Simplify writing asyn code, specifically Promises

```
async function first () {
    return "First";
}
first().then (val => console.log(val));
```

first() → returns promise

await

→ Reserved keywords used inside async func".
→ Pauses the execution of asyn func" & is followed by Promise.
Basically waits for Promise to resolve & then resumes the async's function execution & returns resolved value.
(Similar to yield with generators)

```
async function getMovieData() {
    Consol.log("start");
    var movieData = await $.getJSON(          );
}
```

waits for Promise to be resolved

Doesn't run:

```
    consol.log("All done!");
    consol.log(movieData);
```

```
getMovieData();
```

methods in objects can also be async.

Note: Await assumes that promise is always resolved. So, for error use try catch.

```
try {
    await
}
catch(e) {
    consel.log(e);
}
```

Date
Page

* We can use await with Promise.all( )

5 Object Rest & Spread:

REST var instructor = { first: 'Elie', last: "Shop", numSib: 3 };

var {first, ...data} = instructor

first 'Elie'

data { last: 'Shop', numSib: 3 }

Spread

var instructor2 = {...instructor, first: "Tim", age: 18};

overwrite

Type Script

- Ways to declare a type object

```
interface Point {
readonly x: number;
    y: number;
}

type Point = {
    x: numb;
    y: -
}
```

- ✱ You can specify optional parameters using '?'
- ✱ Extending interface

⇒ interface ColorfulCircle extends Colorful, Circle {}

we can also use 'intersection'

type ColorfulCircle = Colorful & Circle;

- Generics

function doWork<Type>(x: Array<Type>): Type {}

```
interface Box<Type> {
    contents: Type;
}
```

```
type Box<Type> = {
    contents: Type;
}
```

Note: Array is a built-in generic type / interface.

* we now also have Map<K, V>
  Set<T>
  Promise<T>

* Generic classes

```
class GenericNumber<NumType> {
    zero: NumType;
    add: (x: NumType) => NumType;
}
```

* Adding Constraints

```
interface OurConstraint {
    length: number;
}
function func<Type extends OurConstrain>(arg: Type) {
    console.log(arg.length);
}
```

⇒ Special Types

ReturnType < type of func" >

ReadOnlyArray <T>

.keyof

⇒ Classes

```
class xyz {
default [public    x : string;
        protected y : number;
        private z : string;
};
```

get ()
set ()
static

Note: JS have no static classes.

A TS way of handling with "this"

```
class MClass {
    name = "xyz";
    getName (this: MClass) {
        return this.name;
    };
};
```