JAVASCRIPT

- Lightweight
- Cross-platform
- Object oriented

JS can be used

- Client-Side: traditional
- Server-Side: node.js
  - Dynamic effects and interactivity
  - Modern web applications that can we interact with

SCRIPT and STATEMENT

- **Script**; is a serious of instructions that a computer can **follow one by one**.
- **Statement**; each individual instruction or step is known as a statement. Should end with a ";"
  - Each of line of code in GREEN is as statement
  - The PINK curly braces indicate the start end of a code block
  - The code in PURPLE determines which code should run

COMMENTS

- **Multi-Line Comment;**
  - used for descriptions of how the scripts works or to prevent a section of the script from running when testing it
  - PINK
  - /*
- **Single comments;**
  - used for short description of what the code is doing
  - GRAY
  - //

- console.log("hello world"); //writing in the console alert box
- alert("hello world"); //print screen in the POP-UP box
- document.write("hello world"); //HTML

VARIABLE

- var quantity = 3;
  quantity : variable name
  variable value : 3
- Creating variables:

var price=5; var quantity=14; var total=price*quantity;	var price=5, quantity=14; var total=price*quantity;	var price, quantity, total; price=5; quantity=14; total=price*quantity;

- RULES for VARIABLES NAMES
  - It can start with only LETTER, or $ or _
  - No space, no special characters (except $ and _),
  - Case sensitive "A" not equal to "a"
  - If you see the error 'Read only'; change the variable name
  - Clean and meaningful and camel case not use 'ts' use 'toothSize'
  - Variable can not be any of JS reserved words or keywords

DATA TYPES (JS PRIMITIVE)

1. **Numbers**: 1, 1.2
2. **String**: Sequences of characters, used for text (PROMPT is always STRING)
3. **Boolean**: True or False
4. **Undefined**: Data type of a variable which does not have a value yet
5. **Null**: Non-existent

- How to check datatypes?
  - typeof variable_name;

var a = 10 + 'a'; var b = 'ten' + 10; var c;

console.log(a);	//10a	console.log(typeof a);	//string
console.log(b);	//ten10	console.log(typeof b);	//string
console.log(c);	//undefined	console.log(typeof c);	//undefined

PROMPT OPERATOR

- It is used to assign values **during runtime**
- Data Type of Prompt is always STRING
- var name;
  name = prompt ('please enter your name');
  console.log(name);

```
var name, age, address, status;
name = prompt('please enter your name');
age = prompt('please enter your age');
address = prompt('please enter your address');
console.log(name,typeof name);   // STRING  Because PROMPT is always STRING
console.log(age,typeof age);   // STRING
console.log(address,typeof address);   // STRING
```

CONCATENATION

- The action of linking things together.

```
var firstName = 'Eyup';
var lastName = 'Aydin'
var address = 'xxx Grace Ave, Garfield, NJ 07026';
var age = 28
console.log(firstName + ' ' + lastName + ' is ' + age + ' years old. ' + firstName + ' is living at ' + address + '.' )
```

OPERATORS

- Expressions rely on things called operators; to create a single value from one or more values

1. ***Arithmetic Operators***

NAME	OPERATOR	PURPOSE	EXAMPLE	RESULT
Addition	+	Adds Value	10+5	15
Subtraction	-	Subtract Value	10-5	5
Division	/	Divides	10/5	2
Multiplication	*	Multiplies	10*5	50
Increment	++	Adds 1	i=10; i++	11
Decrement	--	Subtract 1	i=10; i--	9
Modulus	%	Divides and remains	10%3	1

Arithmetic Operators Examples;

```
// operator : x + y
// Number + Number = Addition
console.log(2+4);
console.log(20+200);

// number + string = Concatenation
console.log(2+'abi');  // 2abi

// string + boolean = Concatenation
console.log('abi'+true);
console.log(true+'abi');
console.log(false+'abi');

// string + string = Concatenation
console.log('abe'+'abi');

//Operator : x-y
console.log(5-3);
console.log(10-20);
console.log(-10-10);
```

```
// boolean + number = Addition
console.log(false + 1);  // false is 0
console.log(true + 1); //Result is 2 /////
TRUE means 1

//Operator : x*y
console.log(10*10);
console.log(20*-10);

//Operator:x/y
console.log(10/10);
console.log(20/10);
console.log(10*10);
console.log(20/0);
console.log(0/20);

// operator : x%y
console.log(5%3);          //2
console.log(6%2);          //0
```

```
// operator : x++ or ++x
var x = 3;
//y=x++     // Result is x=4 y=3
y=++x;     // Result is x=4 y=4
console.log(x);
console.log(y);
```

```
// operator : x--  or --x
var x=3;
//y=x--;     // Result is x=2 y=3 y first look at x
y=--x;       // Result is x=2 y=2 y firstly check -x
console.log(x);
console.log(y);
```

2. *Assignment Operators*

NAME	SHORTHAND OPERATOR	MEANING
Assignment	x = y	x = y
Addition Ass	x += y	x = x + y
Subtraction Ass	x -= y	x = x - y
Multiplication Ass	x *= y	x = x * y
Division Ass	x /= y	x = x / y
Remainder Ass	x %= y	x = x % y

3. *Comparison Operators*

All comparison operators return **Boolean** (true or false)

DESCRIPTION	OPERATOR	EXAMPLE	RESULT
Equality	==	console.log('1'==1);	TRUE
Inequality	!=	console.log(1!='1');	FALSE
Identity / Strict Equality	===	console.log(1==='1');	FALSE
Non-Identity / Strict Inequality	!==	console.log(1!=='1');	TRUE
Greater than	>	console.log(1>1);	FALSE
Greater than or equal	>=	console.log(1>=1);	TRUE
Less than	<	console.log(1<1);	FALSE
Less than or equal	<=	console.log(1>=1);	TRUE

// Equality
```
console.log(1==1); //TRUE
console.log('1'==1); //TRUE
console.log(0==false); //TRUE
```

//Inequality
```
console.log(1!=2); //TRUE
console.log(1!='1'); //FALSE
console.log(1!=false); //TRUE
```

// strict equality
```
console.log(1===1); //TRUE
console.log(1==='1'); //FALSE
```

// non-strict inequality
```
console.log(1!=='1'); //TRUE values are same, but dataTypes is not equal
console.log(1!==1); //FALSE
```

```
// Greater Than
console.log(1>1); //FALSE
// Greater than or equal
console.log(1>=1);
console.log(1>='1');  // TRUE
// less Than
console.log(1<1);
// less than or equal
console.log(1<=1);
console.log('1'<='1'); //TRUE
```

4. ***Logical Operators***

OPERATOR	DESCRIPTION	EXAMPLE
&&	AND	(x<10 && y>1)
||	OR	(x=5 || y==5)
!	NOT	!(x==y)

&&	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

||	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Expression	Returned Value (!)
False	True
True	False

Logical Operator Examples;
```
// logical &&
console.log(true&&true);   // TRUE
console.log(true&&false);   // FALSE
console.log(false&&(3==4));    // FALSE

// logical OR ||
console.log(true||true);   // TRUE
console.log(true||false);   // TRUE
console.log(false||(3==4));   // FALSE
```

CONTROL FLOW STATEMENTS

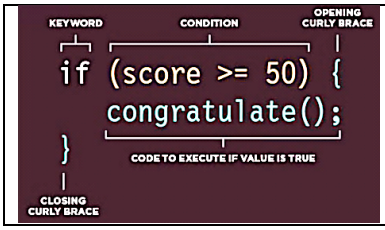
1. **Selection Statements:**
   - if, if - else, if-else if - else
   - switch case
2. **Looping Statements:**
   - while, do-while
   - for
3. **Branching Statements:**
   - break
   - continue

1. *Selection Statements:*

- **if, if - else, if - else if – else**

a) if

- evaluates a condition. If the condition evaluates to TRUE, any statements in the subsequent code block are executed.

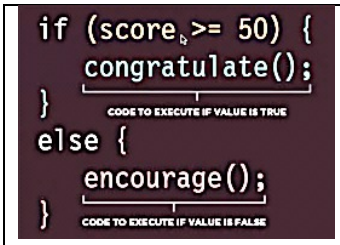



```
var numWaterMellon = 10;
if (numWaterMellon>=20){
   console.log("I have 20 and more
watermellon.");   // result is Nothing
}
```

```
var firstName = "Mike";     // we can do it also with prompt
var status = "Married";
if (status == "Married"){
   console.log(firstName + " is " + status);
}
if (num1>num2 || num1<num2){
   console.log("These two numbers are not equal");
}
```

b) if else

- checks a condition. If it resolves to **TRUE** the first code blocks is executed. If the condition resolves to **FALSE**, the second code block is run instead.




```
var name = "Mike";
var status = "Married";
if (status="Married"){
   console.log("Mike is married");
}  else{
   console.log("Mike is single");
}          // RESULT: Mike is married
```

```
var numA, lotsA;
lotsA = false;
numA=prompt("enter your apple!");
if (numA>=20){
   console.log("I have more than 20 apple");
   lotsA=true;
   console.log("Good Job");
} else {
   console.log("I need more apple")
}
```

c) if … else if … else;

- nested to create an ELSE IF clause. It is used to make decision among several alternatives.

if (condition1){ statement1; }else if(condition2){ statement2; }else{ statement3; }	//1-Monday, 2-Tuesday, 3-Wednesday, 4-Thursday, var day = parseInt(prompt("Enter your number")); if (day == 1){ console.log("Mon"); }else if (day == 2){ console.log("Tue"); }else if (day == 3){ console.log("Wed"); }else if (day == 4){ console.log("Thur"); }else{ console.log("Not a valid day"); }

❖ Nested IF Statements

```
var expectedUserName, actualUserName;
actualUserName=prompt("Please enter your username!");
expectedUserName="abc";
if (actualUserName!=""){
   if(expectedUserName==actualUserName){
      console.log("correct username");
   }else{
   console.log("incorrect password");
   }
}else{
   console.log("username cannot be empty");
}
```

d) Switch Statements;

- is used to compare the value of a variable with multiple values and execute some statements based on the ***match***.

```
switch (level1) {
   case "one":
        title = "Level 1";
        break;
   case "two":
        title = "Level 2";
        break;
   case "three":
        title = "Level 3";
        break;
   default:
        title = "Test";
        break;
}
```

2. ***LOOP ing Statements***

Check a condition.

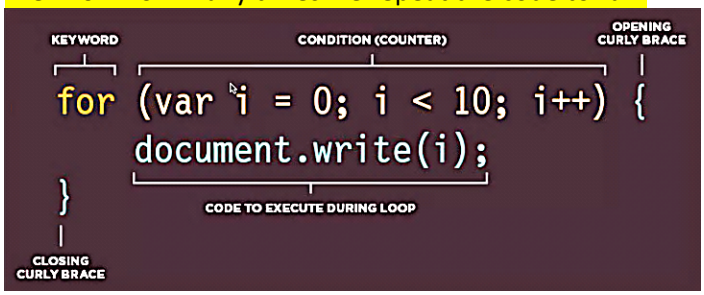
- If it returns TRUE, a code block will run.
- Then the condition will be checked again and if it is still returns TRUE, the code will run again.
- If repeats until the condition return FALSE.
  - FOR
  - WHILE
  - DO WHILE

a) FOR

If you need to run code a specific number of times, use a for loop.
In a for loop, the condition is usually a counter which is used to tell how many times the loop should run.

What is the difference between the for loops and others?

We know how many times we repeat the code to run.



Initialization **var i = 0;**	**It is executed once, as the loop begins**
Condition / Termination **i<10;**	When the terminate expression **evaluates to FALSE, the loop terminates**
increment / Update **i++**	The increment expression is **invoked after each iteration through the loop** / it is also possible to decrement

❖ Nested For Loop Statement

```
for (var i=0; i<5; i++){
   console.log("outer loop at state: " + i);
   for(var p=0; p<2; p++){
      console.log("inner loop at state: " + p);
      if(p==1){
         console.log("\n");
      }
   }
}
```

b) while loop

```
while(condition){
        statement;
}
```

If you don't know how many repeat the condition; (if you know it, it is FOR LOOP)

- while loop repeatedly executes same set of statements as long as condition is TRUE
- condition is checked at loop start
- the code is repeated in a loop is called the body of the loop.
- Each repetition of the loop body is called an iteration of the loop

If condition is false WHILE is doing NOTHING

c) do … while loop

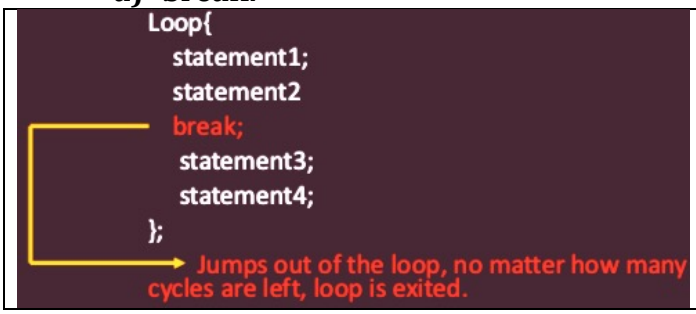
```
do {
        statements;
}while(condition)
```

- do … while loop repeatedly executes same set of statements while(condition) is TRUE
- the statement is executed once initially the Condition is checked at the loop end
- code in loop body will execute at least once

If condition is FALSE, do statements

3. *Branching Statements*

a) break:

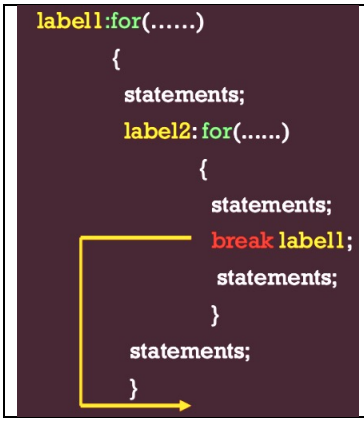


This keyword causes the termination of the loop and tells the interpreter to go onto the next statement of code outside of the loop. //Break always get out of the loop.

b) continue:

The continue statement skips the current iteration of the loop and jumps to the next iteration // Continue always skip the statements and goes the condition…

❖ Labeled Statements



It is possible to break and continue outer loops when dealing with nested loops. In these cases, the loops must be annotated with some label, and the label must be passed to the break/continue statement.

Summary of Loops

- Conditional statements allow your code to make decisions about what to do next
- Comparison operators (==,===,!=.!==,<,>,<=,=>) are used to compare two operands
- Logical operators allow you to combine more than one set of comparison operators.
- if...else statements allow you to run one set of code if a condition is true, and another if it is false
- Switch statement allow you to compare a value against possible outcomes
- There are three types of loop: for, while, and do...while. Each repeats a set of statements.

ARRAY

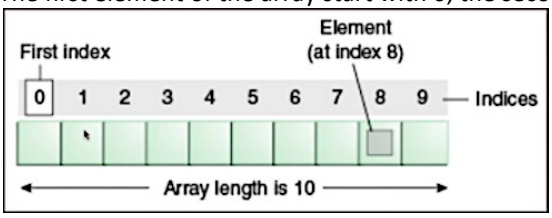
- Is a special type of variable that can hold more than one value. (Array is a special kind of OBJECT)

CREATING AN ARRAY

```
var scores=new Array();          → The scores array is empty, holds no element
var scores=Array(10);            → The scores array with initial size.
var scores=new Array(9,10,8,7);  → The scores array with initial elements.
var colors=["red", "green", "blue"];  → Colors array with three elements.
var emptyArray=[];               → Creating an empty array.
```

ACCESSING ARRAY ELEMENTS

- You can access elements of an array by using the brackets [].
- The first element of the array start with 0, the second element starts with 1, and so on.




```
var cars=["Honda", "ford", "Toyota"];
console.log(cars[0]);    //    Honda
console.log(cars[1]);    //    ford
console.log(cars[2]);    //    Toyota
```

THE SIZE of AN ARRAY

- The array uses the length property to store the current number of elements it holds

```
var cars=["Honda", "ford", "Toyota"];
console.log(cars.length);  // 3
```

BASIC OPERATIONS on ARRAYS

- to check is a variable is an array, you use Array.isArray() method.

```
var cars=["Honda", "ford", "Toyota"];
console.log(Array.isArray(cars));    //TRUE
```

- When you call the toString() method of an Array, you get a string represented as a comma-separated list of elements

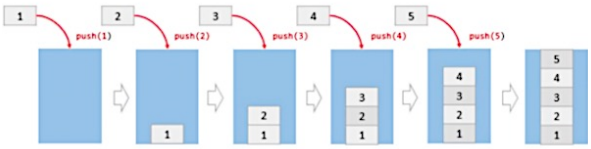
```
var cars=["Honda", "ford", "Toyota"];
console.log(cars.toString());        //Honda,ford,Toyota
```

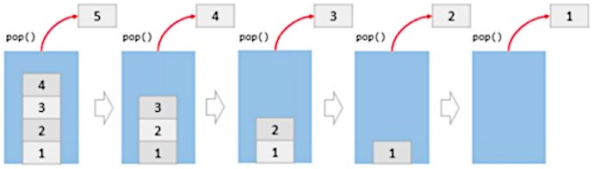
JS Stack Using an Array

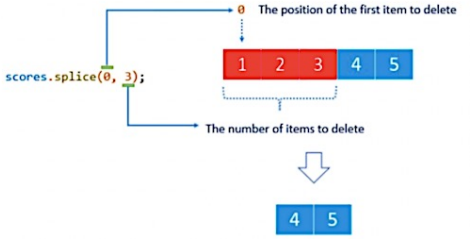
- A stack is a data structure that holds a list of elements. A stack works based on the Last In, First Out, meaning that the most recently added the element is the first one to remove.
- JS Array type provides the push() and pop() methods that allow you to use an array as a stack.

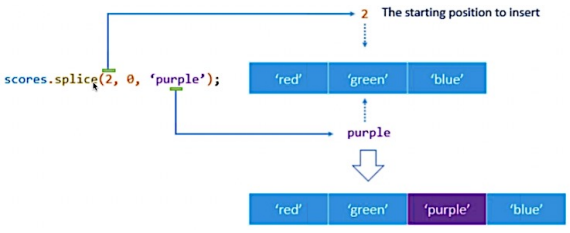
Methods:

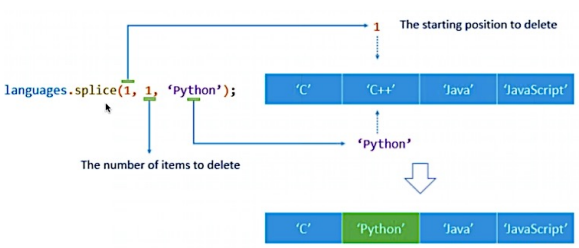
<table>
<tr>
<td>.push(…items)

allows you to add one or more elements to the end of the array</td>
<td colspan="2"> 
</td>
<td>var stack=[];
stack.push(1);
console.log(stack); // [1]
stack.push(2);
console.log(stack); // [1,2]
stack.push(3);
console.log(stack); // [1,2,3]
stack.push(4);
console.log(stack); // [1,2,3,4]
stack.push(5);
console.log(stack); // [1,2,3,4,5]</td>
</tr>
<tr>
<td>.pop( )

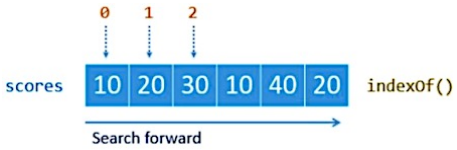
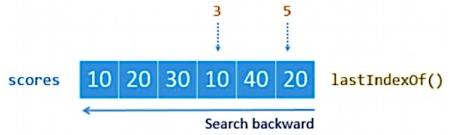
removes the element at the end of the array and returns the element to the caller.</td>
<td colspan="2"> 
</td>
<td>console.log(stack.pop()); //5
console.log(stack); // [1,2,3,4]
console.log(stack.pop()); //4
console.log(stack); // [1,2,3]
console.log(stack.pop()); //3
console.log(stack); // [1,2]
console.log(stack.pop()); //2
console.log(stack); // [1]
console.log(stack.pop()); //1
console.log(stack); // []</td>
</tr>
<tr>
<td>.shift( )

removes the first element from an array and returns that removed element.</td>
<td colspan="3">var myFish = ["Angel", "Clown", "Mandarin", "Surgean"];
var shifted=myFish.shift();
console.log(shifted); //Angel
console.log(myFish); //(3) ["Clown", "Mandarin", "Surgean"]
for (var i=0;i<myFish.length;i++){
console.log(myFish[i]);
}</td>
</tr>
<tr>
<td rowspan="3">.splice( )</td>
<td> 
</td>
<td>DELETES element in an array, you pass TWO ARGUMENTS to the splice method.

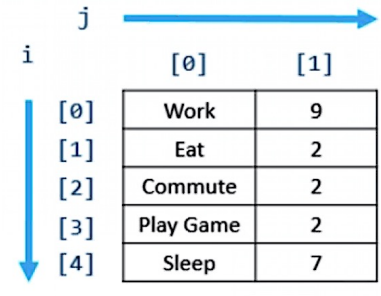
DELETE
.splice(1st, 2nd);
1st → start to delete
2nd → how many</td>
</tr>
<tr>
<td> 
</td>
<td>insert one /more element into an array by passing THREE or more arguments to the splice() method with the second argument is ZERO.

ADD ing without DELETING
.splice(1st, 2nd, 3rd);
1st → where we add
2nd → must be 0
3rd → what we add</td>
</tr>
<tr>
<td> 
</td>
<td>You can insert new elements into an array while deleting existing elements simultaneously by using splice() method.
Second is NOT ZERO → CHANGE the VALUE

ADD ing WITH DELETING
.splice(1st, 2nd, 3rd);
1st → where we put
2nd → How many items you delete
3rd → what we add</td>
</tr>
</table>

.slice( ) returns a shallow copy of a portion of an array into a new array selected from begin to end (END NOT INCLUDED). The original array will not be modified.	var animals = ["ant", "bison", "camel", "duck", "elephant"]; console.log(animals.slice(2)); //["camel", "duck", "elephant"]; console.log(animals.slice(2,4)); //["camel", "duck"] console.log(animals.slice(1,5)); //["bison", "camel", "duck", "elephant"]; console.log(animals.slice(1,2)); //["bison"] console.log(animals.slice(-1)); //["elephant"]
.indexOf( ) returns the first index at which a given element can be found in the array OR -1 if it is not present	 var scores = [10,20,30,10,40,20]; console.log(scores.indexOf(30)); // 2 console.log(scores.indexOf(50)); // -1 console.log(scores.indexOf(20)); // 1 console.log(scores.indexOf(10,2)); // 3 → go [2] check rest, find 10 console.log(scores.indexOf(30,3));// -1 → go [3] check rest, find 30 console.log(scores.indexOf(20,-1));// 5 → (fromIndex = 6 + (-1) =5) console.log(scores.indexOf(20,-5));// 1 → (fromIndex = 6 + (-5) =1)
.lastIndexOf( ) returns the last index at which a given element can be found in the array OR -1 if it is not present	 var scores = [10,20,30,10,40,20]; console.log(scores.lastIndexOf(10)); // 3 console.log(scores.lastIndexOf(20)); // 5 console.log(scores.lastIndexOf(50)); // -1
.sort( ) sorts the elements of an array in place and **returns the array.**	var months = ["March","Jan","Feb","Dec"]; months. sort(); console.log(months); //["Dec", "Feb", "Jan", "March"] var array1 = [1,30,4,21]; array1.sort(); console.log(array1); //[1, 21, 30, 4] var mixArray = ["cat", "Ant",3,"elephant","Bee",67]; mixArray.sort(); console.log(mixArray); //[3, 67, "Ant", "Bee", "cat", "elephant"]
.reverse( ) it doesn't only return the array in reverse order, **it also reverses the order of the array itself.**	var arr1 = "john".split(''); //["j", "o", "h", "n"] var arr2 = arr1.reverse(); //["n", "h", "o", "j"] var arr3 = "jones".split(''); //["j", "o", "n", "e", "s"] arr2.push(arr3); // 5 console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1)); //array 1: length=5 last=j,o,n,e,s console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1)); //array 2: length=5 last=j,o,n,e,s

MULTIDIMENSIONAL ARRAY

Create	var activities = [["Work", 9], ["Eat", 2] ["Commute", 2] ["Play Game", 2] ["Sleep", 7]];	Work 9 activities [0]; Eat 2 activities [1]; Commute 2 activities [2]; Play Game 2 activities [3]; Sleep 7 activities [4];
Access	console.log(activities[0][1]); // 9 console.log(activities[3][0]); // Play Game	
Iterate		// loop the outer array for (var i=0; i<activities.length; i++){ //get the size of the inner array var innerArrayLength = activities[i].length; //loop the inner array for (var j=0; j<innerArrayLength; j++){ console.log("["+i+","+j+"] ="+activities[i][j]) } }

FUNCTIONS

Functions, let you group a series of statements together to perform a specific task. If different parts of a script repeat the same task, you can reuse the function (rather than repeating the same set of statements).

1

```
var dogName="rover";
var dogWeight="23";

if (dogWeight>20){
   console.log(dogName+" says WOOF WOOF");
}else{
   console.log(dogName+" says woof woof");
}
```

2

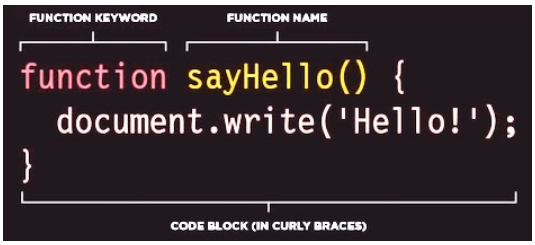
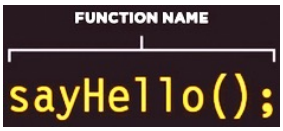
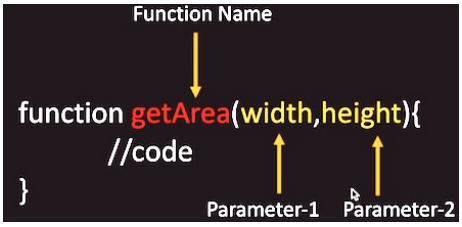
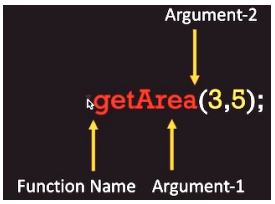
```
var dogName="spot";
var dogWeight="13";

if (dogWeight>20){
   console.log(dogName+" says WOOF WOOF");
}else{
   console.log(dogName+" says woof woof");
}
```

3

```
var dogName="spike";
var dogWeight="53";

if (dogWeight>20){
   console.log(dogName+" says WOOF WOOF");
}else{
   console.log(dogName+" says woof woof");
}
```

4

```
var dogName="lady";
var dogWeight="17";

if (dogWeight>20){
   console.log(dogName+" says WOOF WOOF");
}else{
   console.log(dogName+" says woof woof");
}
```

Declaring a FUNCTION	You give it a name and then write the statements needed to achieve its task inside the curly braces.	
Calling a FUNCTION	Having declared the function, you can then execute all of the statements between its curly braces with just one line of code.	
Declaring Functions That Need Information	Sometimes a function needs specific information to perform its task. In such cases, when you declare the function you give it PARAMETERs. Inside the function, the parameters act like variables.	
Calling Functions That Need Information	When you call a function that has parameters, you specify the values it should use in the parentheses that follow its name. The values are called arguments, and they can be provided as values or as variables.	
	`function abc(width, height){` `  var area = width * height;` `  console.log(area);  //6` `  console.log(width);  //2` `  console.log(height);  //3` `}` `abc(2,3);`	`function bark(name, weight){` `  if (weight>20){` `  console.log(name+" says WOOF WOOF");` `  }else{` `  console.log(name+" says woof woof");` `  }` `}` `bark("rover",23);` `bark("spot",13);` `bark("lady",17);`

JavaScript is pass-by-value

- JavaScript passes arguments to a function using pass-by-value. Each arguments are copied into the parameter variable.

1

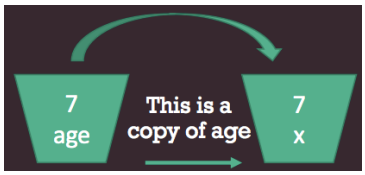
```
var age = 7;
```

2

```
function addOne(x){
      x=x+1;
}
```

3

```
addOne(age);
```



4

```
function addOne(x){
      x=x+1;
}
```

We are incrementing x

Age does not change even if x does

x has been incremented addOne

Age still 7

What will be the output of these functions?

```
function makeTea(cups,tea){
        console.log("Brewing" + cups + " cup of " + tea);
}
```

1- makeTea(3); // Brewing 3 cup of undefined

2- makeTea(3, "Early Grey", "HeyMa", 42); // Brewing 3 cup of Early Grey

How to return a value from function?

- A function can return a value by using the return statement followed by an expression or a value.

```
function calculateArea(width, height){
        var area = width*height;
        return area;                          //area is local variable
}
var wallOne = calculateArea(3,5);
```

JavaScript function overloading

- In other programming languages, it is possible to have two or more functions that share the same name as long as their signature are different are different. However you can not do it in JS.
- NOT: JS consider the Last function!!!

1

```
function addTen(a){
   return a+10;
}

function addTen(a,b){
   return a+b+10;
}

var result =addTen(100);
console.log(result);        //NaN
```

2

```
function addTen(a,b){
   return a+b+10;
}

function addTen(a,b){
   return a+10;
}

var result =addTen(100);
console.log(result);          //110
```

JavaScript function hosting

- In JS, it is possible to use a function before it is declared.

```
showMe();
function showMe(){
    console.log("an hosting example");
}
```

1. In the compilation phase, JS engine moves all function declarations (also variable declarations) to the top of its current context.

```
function showMe(){
    console.log("an hosting example");
}
showMe();
```

2. In the execution phase, JS engine just executes the code.

How to return multiple values from function?

- Functions can return more than one value using an array.

```
function getSize(width, height, depth){
        var area = width*height;
        var volume = width * height * depth;
        var sizes = [area, volume];
        return sizes;
}
var areaOne = getSize(3,2,3)[0];
var volumeOne = getSize(3,2,3)[1];
```

Global and Local Variables

- When a variable is created outside of a function, then it can be used anywhere within the script. It is called a GLOBAL VARIABLE and has GLOBAL SCOPE.
- When a variable is created inside of a function using the var keyword, it can only be used in that function. It is called a LOCAL VARIABLE or function-level variable. It is said to have LOCAL SCOPE or function level scope. It can not be accessed outside of the function in which it was declared.

```
var avatar;
var levelThreshold=1000;
```

These are global variables. They are accessible everywhere in the code.

```
function getScore(points){
    var score;
    var i=0;
```

The points, score, and i var are all declared within a function and they are local variables.

```
    while(i<levelThreshold){
        i=i+1;
    } return score;
}
```

Even if we use levelThreshold inside the function, it is global because it is declared outside the function.

Function Statements vs Expression

	Statement	Expression	
1	if (x===5){ // do something }	3+4; var x = 3;	
2	function someFun(par){ // code }	var someFun = function(par){ // code }	RESULT is SAME

JS STRING MANIPULATION METHODS

- JS string stores a series of characters like “Java Script”.
- A string can be any text inside double or single quotes:

  var string = "Java Script"; var string = ‘Java Script’;
- String indexes are zero-based: The first character is in position 0, the second one is 1 and so on

METHOD	DEFINITION	EXAMPLE
.length	- It returns count of total number of characters. - For an empty, length is 0.	var string = "Java Script"; var x = ""; console.log(string.length); //11 console.log(x.length); //0
.charAt( )	- returns a char value at the given index number	var string = "Java Script"; console.log(string.charAt(2)); //v console.log(string.charAt(4)); // console.log(string.charAt(9)); //p
.concat( )	- Combines specified string at the end of this string. It returns combined string. It is like appending another string.	var s1 = "Hello"; var s2 = "World"; var s3 =s1.concat(s2); console.log(s3); // Hello World
.includes( )	- Determines whether one string may be found within another string, returning **true or false**	var str = "Hello Mike, welcome! " var n = str.includes("Mike"); console.log(n); //**true**
.indexOf( )	- returns the index within the calling String of the first occurrence of the specified value, starting from Index. Returns -1 if the value is not found.	var str = "abreadjambread"; var n=str.indexOf("bread"); //1 **var m=str.indexOf("bread",n+5); //9**
.replace( )	- returns a string replacing all the old char or CharSequence to new char or CharSequence - The original string will remain unchanged	var str = "Ebay is a very good website"; var replaced = str.replace("Ebay","Bestbuy"); console.log(replaced);
.search( )	- searches a string for a specified value, and returns the position of the match	var str = "I want to be number 1 test automation developer"; console.log(str.search("automation"); //27 console.log(str.search("1"); //20
.slice( )	- extracts a section of a string and returns it as a new string.	var str = "The morning is upon us"; console.log(str.slice(1,8); // he morn console.log(str.slice(4,-2); //morning is upon u console.log(str.slice(12); // is upon us
.split( )	- is used to split a string into an array of substrings, and returns the new array. - Delimeter for the split function. It can be a space or any character and JS will look for that inside the string and split.	var str = "It will display the array of the size"; var arr= str.split(" "); console.log(arr); //["It", "will", "display", "the", "array", "of", "the", "size"]
.substr( )	Syntax: string.substr(start,length); START: Required. The position where to start the extraction. First character is at index 0 If start is + && >=string.length, substr() returns an empty string. If start is -, it uses it as a character index from the end If start is - || >=string.length, start is set to 0 LENGTH: **How many times it repeats**	var str = "JavaScript"; console.log(str.substr(0,1)); // J -> 0th index console.log(str.substr(1,0)); // " " -> 1st inx console.log(str.substr(-1,1)); //t ->start fr end console.log(str.substr(1,-1)); // " " console.log(str.substr(-3)); //ipt ->-3rd index console.log(str.substr(1)); //avaScript -> 1st console.log(str.substr(-20,2)); // Ja ->starts 0 console.log(str.substr(20,2)); // " "
.substring( )	Syntax : string.substring(indexStart,indexEnd) - not including indexEnd. -if indexStart = indexEnd =>empty -if indexStart > indexEnd =>**SWAP**	var str="Hello World!"; console.log(str.substr(0,1); //H --go to 0, dont get 1 console.log(str.substr(1,0); //H --1>0 swap! (0,1) console.log(str.substr(4); //o world! --no end index, console.log(str.substr(4,7); //o w
.toLowerCase( )	- returns the calling string value converted to lower case	var str="JavaScript"; console.log(str.toLowerCase(); //javascript
.toUpperCase()	- returns the calling string value converted to upper case	var str="JavaScript"; console.log(str.toUpperCase(); //JAVASCRIPT
.trim( )	- removes whitespace from both sides of a string	var str = " Hello World! "; var btr = " Hel lo W orld! "; console.log(str.trim()); //Hello World! console.log(btr.trim()); //Hel lo W orld!

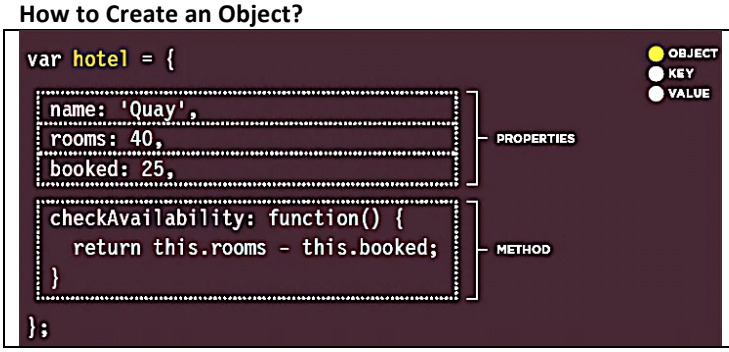
OBJECTS

- Object is a collection of properties and methods/functions.

Object Property: Key + Value

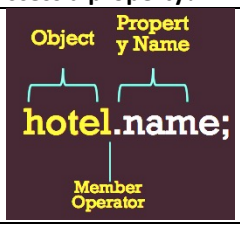
PROPERTIES	KEY	VALUE
	name	string
	rooms	number
	booked	number
	gym	Boolean
	roomtypes	array
METHODS	checkavailability	function

How to Create an Object?



- Make sure to enclose your object in curly braces:
- Separate the property name and property value with a colon:
- Separate each property name and value pair with a comma:
  Do not use a comma after the last property value:

How to access a property?



- To access a property in an object, start with the object name, follow it with a period and then use the property name.

Updating an Object

- To update the value of properties, use dot notation or square brackets. To delete a property, use the delete keyword.
- If the object does not have the property you are trying to update, it will be added to the object.

fiat["name"]="bmw" delete fiat.make;
fiat.name="honda"

Adding behavior to object

```
var car = {
      make: "Honda",
      model: "Honda",
      year: 2000,
      color: "Red",
      mileage: 75000,

      drive : function(){
            console.log("I am driving honda"),
      }
};
car.drive(); //I am driving Honda
```

How ".this" keyword works?

-this reference to my object name (like car here)

```
var car = {
    make: "Honda",
    model: "Honda",
    year: 2000,
    color: "Red",
    mileage: 75000,
    started: false,
    start: function(){
        this.started=true;
    },
    stop: function(){
        this.started=false;
    },
    drive : function(){
        if (this.started){
            console.log("it is running"),
        }else{
            console.log("you need to start the engine first"),
        }
    }
};
```

```
car.drive(); // you need to start the engine first
car.start();
car.drive(); // it is running
car.stop();
car.drive(); // you need to start the engine first
```

for … in statement

- iterates over the properties of an object. For each distinct property, statements can be executed.

```
for (variable in function name){
    //code
}
```

```
var car = {
    make: "Honda",
    model: "Honda",
    year: 2000,
    color: "Red",
    mileage: 75000
};
for (x in car){
    console.log(x+ ":" + car[x]);
}
```

Creating an Object: CONSTRUCTOR NOTATION

- The new keyword and the object constructor create a blank object. You can then add properties and methods to the object

```
var hotel = new Object();

hotel.name = 'Quay';
hotel.rooms = 40;
hotel.booked = 25;

hotel.checkAvailability = function() {
  return this.rooms - this.booked;
};
```

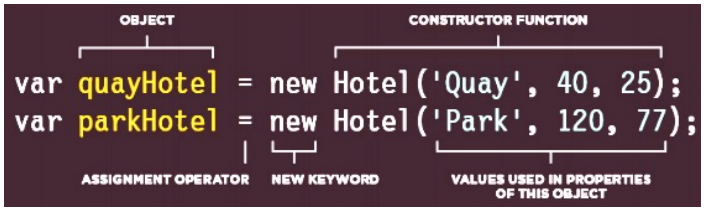
OBJECT
KEY
VALUE
PROPERTIES
METHOD

var obj1=new Object(); obj1.name="John"; obj1.lastName="Mike"; obj1.age=20;	var obj3=new Object(); obj3.name="Kevin"; obj3.lastName="Eric"; obj3.age=30;
var obj2=new Object(); obj2.name="Mike"; obj2.lastName="John"; obj2.age=25;	var obj4=new Object(); obj4.name="Eric"; obj4.lastName="Kevin"; obj4.age=32;

Object Constructor Function

- Object constructors can use a function as a template for creating objects. First, create the template with the object's properties and methods.
- this keyword is used instead of the object name to indicate that the property or method belongs to the object that this function creates

Creating Many Objects



Adding Property and Method to a Object

```
//Adding a Property to an Object
        myFather.nationality = "English";
//Adding a Method to an Object
        myFather.name = function (){
                return this.firstName + ""+ this.lastName;
        };
```

Adding Property and Method to a Constructor

//Adding a Property to a Constructor

```
function Person(first,last,age,eyeColor){
    this.firstName=first;
    this.lastName=last;
    this.age=age;
    this.nationality="English";
}
```

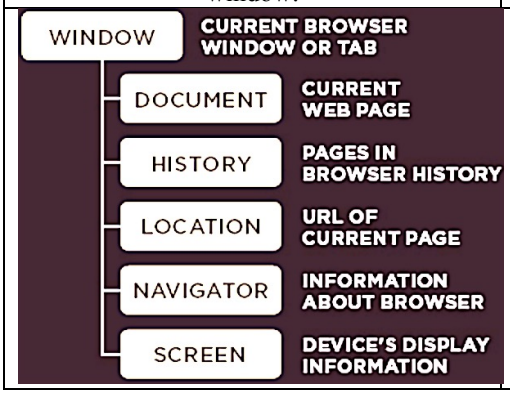
//Adding a Method to a Constructor

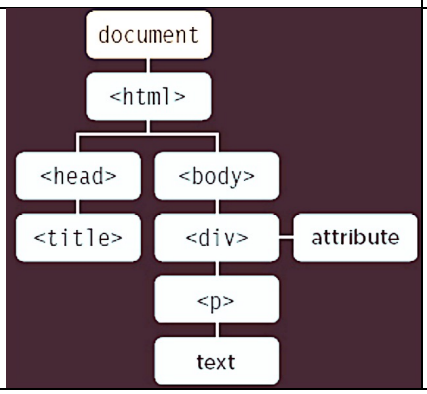
```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.name = function() {return this.firstName + " " + this.lastName;};
}
```

Built-in JavaScript Constructors

- JavaScript has built-in constructors for native objects.
- There are three sets of built-in objects each offer a different range of tools that help you write scripts for web pages.

Browser Object Model	Document Object Model	Global JS Objects
creates a model of the browser tab or window.	creates a model of the current page.	do not form a single model. They are group of individual objects that relate to different parts of the JavaScript.





Represent basic data types	Help deal with real world concepts
STRING	DATE
NUMBER	MATH
BOOLEAN	REGEX

Number Object

Method	Description	Example
isNaN( )	Checks if the value is not a number	console.log(isNaN("23"));//false console.log(isNaN("abc23"));//true console.log(isNaN(23));//false console.log(isNaN(" "));//false
.toFixed( )	Rounds to specified number of decimal places(returns a string)	var a = 213.73145; console.log(a.toFixed(); //214 var b=213.73645; console.log(b.toFixed(2); //213.74
.toPrecision( )	Rounds to total number of places(returns a string)	var a = 13.3714; console.log(a.toPrecision(); //13.3714 console.log(a.toPrecision(2); //13 console.log(a.toPrecision(3); //13.4 var a = 0.001658; console.log(a.toPrecision(); //0.001658 console.log(a.toPrecision(2); //0.0017
.toExponential( )	Returns a string representing the number in exponential notation	var a = 123456; console.log(a.toExponential()); //1.23456e+5 var a = 123.45; console.log(a.toExponential()); //1.23456e+2

.toString( ) Method

- Convert a number to a String

```
var num = 15;
var str = num.toString();
console.log(typeof(num)); //number
console.log(typeof(str)); //string
```

Math Object

- The Math object has properties and methods for mathematical constants and functions.

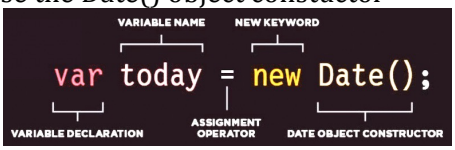
Property/Method	Description	Example
Math.PI	Returns pi (3.14159265359)	Math.PI
Math.round( )	Rounds number to the nearest integer	console.log(Math.round(2.50)); //3 console.log(Math.round(2.49)); //2 console.log(Math.round(-2.50)); //-2 console.log(Math.round(-2.49)); //-2 console.log(Math.round(-2.51)); //-3
Math.sqrt(n)	Returns square root of positive number	console.log(Math.sqrt(64)); //8 console.log(Math.sqrt(0)); //0 console.log(Math.sqrt(-9)); //0
Math.ceil( )	Rounds number **UP** to the nearest integer	console.log(Math.ceil(2.51)); //3 console.log(Math.ceil(2.50)); //3 console.log(Math.ceil(2.49)); //3 console.log(Math.ceil(-2.50)); //-2 console.log(Math.ceil(-2.49)); //-2 console.log(Math.ceil(-2.51)); //-2
Math.floor( )	Rounds number **DOWN** to the nearest integer	console.log(Math.floor(2.51)); //2 console.log(Math.floor(2.50)); //2 console.log(Math.floor(2.49)); //2 console.log(Math.floor(-2.50)); //-3 console.log(Math.floor(-2.49)); //-3 console.log(Math.floor(-2.51)); //-3
Math.random( )	Generates a random number between 0 (inclusive) and 1 (not inclusive)	console.log(Math.random()); //returns a random # between 0 & 1 console.log(Math.random()*10+1); // between 1 & 10 console.log(Math.random()*100+1); // between 1 & 100

Little Extra;

console.log(**Math.pow**(8,2)); //64
console.log(**Math.abs**(-4.7)); //4.7
Math.min(0, 150, 30, 20, -8, -200); // returns -200
Math.max(0, 150, 30, 20, -8, -200); // returns 150

Creating an Instance of The Date Object

- In order to work with dates, you create an instance of the Date object. To create a Date object, use the Date() object constuctor



How to Initiate a Date?

Create new date object		
with the current date and time	var d = new Date();	var d = new Date(); // Thu Aug 16 2018 10:20:28 GMT-0400
as zero time plus the number	var d = new Date(value);	var d = new Date(100000000000); // Thu Aug 16 2018 10:20:28 GMT-0400
from the specified date and time	var d = new Date(dateString);	var d = new Date("October 13,2018 11:13:00"); // Sat Oct 13 2018 11:13:00 GMT-0400
from the specified date and time	var d = new Date(year, month, day, hours, minutes,seconds,miliseconds);	var d = new Date(99,**5**,24,11,33,30,0); // Thu **Jun** 24 1999 11:33:30 GMT-0400

// if no arguments are provided, the constructor creates a JavaScript Date Object for the current date and time according to system settings for timezone offset.
//If at least two arguments are supplied, missing arguments are either set to 1(if day is missing) or 0 for all others

Date Formats

Type	Example	Output
ISO Date	"2015-03-25"	Tue Mar 24 2015 20:00:00 GMT-0400
Short Date	"03/25/2015"	Wed Mar 25 2015 00:00:00 GMT-0400 (Eastern Daylight Time)
Long Date	"Mar 25 2015" or "25 Mar 2015"	
Full Date	"Wednesday March 25 2015"	

GET DATE METHODS

Method	Description	var today = new Date();
getFullYear()	**(yyyy)**	var year = today.getFullYear(); console.log(year); //2018
getMonth()	**(0-11)**	console.log(today.getMonth(); // 7
getDate()	**(1-31)**	console.log(today.getDate(); // 10
getHours()	**(0-23)**	console.log(today.getHours(); // 16
getMinutes()	**(0-59)**	console.log(today.getMinutes(); //
getSeconds()	**(0-59)**	console.log(today.getSeconds(); //
getMiliseconds()	**(0-999)**	console.log(today.getMiliseconds(); //
getTime()	**(miliseconds since January 1,1970)**	console.log(today.getTime(); //
getDay()	**(0-6) → 0 is SUNDAY**	console.log(today.getDay(); //

SET DATE METHODS

	Description	var theBigDay = new Date(1962,6,7); //1962-07-07
setDate()	Set the DAY as a number (1-31)	theBigDay.setDate(24); console.log(theBigDay); // 1962-07-24
setFullYear()	Set the YEAR	console.log(theBigDay.setFullYear(2018,11,3)); // Mon Dec 03 2018 00:00:00 GMT-0500
setMonth()	Set the MONTH (0-11)	console.log(theBigDay.setMonth(10));
setHours()	Set the HOUR (0-23)	console.log(theBigDay.setHours(7));
setMinutes()	Set the MINUTES (0-59)	console.log(theBigDay.setMinutes(7));
setSeconds()	Set the SECONDS (0-59)	console.log(theBigDay.setSeconds(10));
setMiliseconds()	Set the miliseconds (0-999)	console.log(theBigDay.setFullYear());
setTime()	Set the TIME (milliseconds since January 1, 1970)	var event1 = new Date("July 1,1999"); var event2 = new Date( ); event2.setTime(event1.getTime()); console.log(event1); console.log(event2);
setDay()	**(0-6) → 0 is SUNDAY**	console.log(theBigDay.setFullYear());

OBJECT ORIENTED PROGRAMING CONCEPTS in JAVA

OOP

- An object-based application in Java is based on
  1. Declaring classes, (like template)
  2. Creating objects from them and
  3. Interacting between these objects.

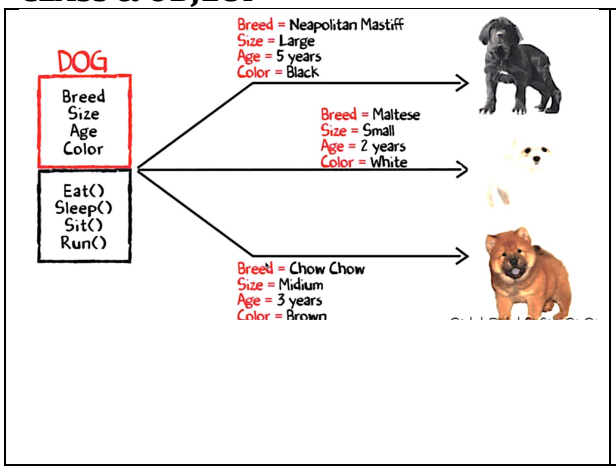
CLASS

- A class is a blueprint or prototype (it is an idea, definition, description) that defines the variables and the methods (functions) common to all objects of a certain kind.

```java
public class Employee {
  public String empName; //Employee name
  public double hourly_rate, hours_worked;
  public double getWeeklyHours() {
   return (hourly_rate * hours_worked);
  }
} //end of class
```

Attributes → **PROPERTIES**	Behaviors → **METHODS**
name	walk
weight	run
height	jump
gender	speak
age	sleep
Person has a name, but **it doesn't say what the name is.** Because the class is **just a description** of something	

CLASS & OBJECT



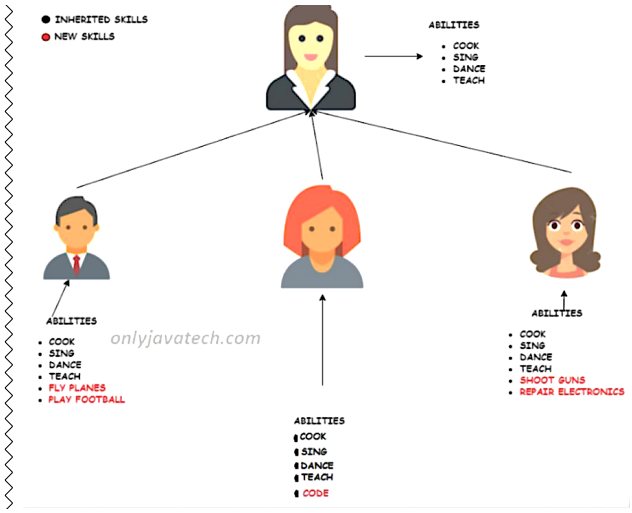

```java
class Dog{

    //variables
    String bread;
    String size;
    int age;
    String color;

    //method
    public String getInfo(){
        return("Breed is: " + breed + "Size is: " + size);
    }
}

public class Execute{
    public static void main(String[] arg){
        Dog maltese = new Dog();
        maltese.breed = "Maltese";
        maltese.size = "Small";
        maltese.age = 2;
        maltese.color = "white";
        System.out.println(maltese.getInfo());
    }
}
```

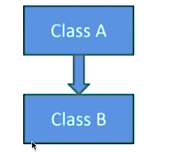
OOP CONCEPTS IN JAVA

1- Inheritance (JS)

- Computer programs are designed in such a way where everything is an object that interacts with one another. (*It is code reusability*)
- Inheritance is one such concept where the properties of one class can be inherited by the other. It helps to reuse the code and establish a relationship between different classes.

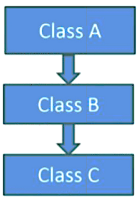
- Parent class (Super or Base Class)
- Child Class (Subclass)





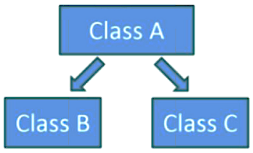
Single Inheritance

```
Class A
{
---
}
Class B extends A {
---
}
```



Multilevel Inheritance

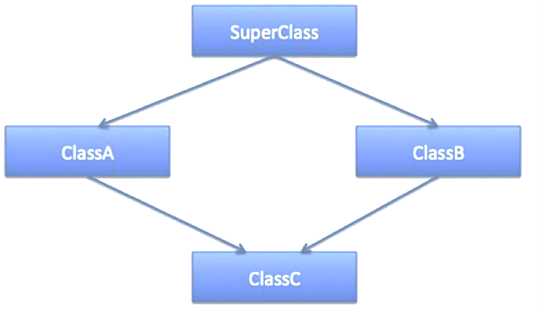
```
Class A{
---
}
Class B extends A{
---
}
Class C extends B{
---
}
```



Hierarchical Inheritance

```
Class A{
---
}
Class B extends A{
---
}
Class C extends A{
---
}
```

Class B has everything class A that have.

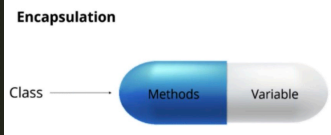


Deadly Diamond of Death

Multiple inheritance is **NOT** supported in Java

2- Encapsulation (JS)

```
public class Person{

    int age;

}

public class Test{

    public static void main(String[] args){

        Person p1 = new Person();
        p1.age = -20;
    }
}
```



It is a mechanism where you bind your data and code together as a single unit. It also means to hide your data in order to make it safe from any modification.

We can achieve encapsulation in Java by;
- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the variables values.

```
public class Person{

    private int age;

    public int getAge(){

        return age;
    }

    public void setAge(int age){

        if(age<18 || age>60){

            //error

        }else{

            this.age = age;

        }
    }
}
public class Test{

    public static void main(String[] args){

        Person p1 = new Person();
        p1.setAge(-20);

    }
}
```

void – your method doesn't return anything

3- Abstraction

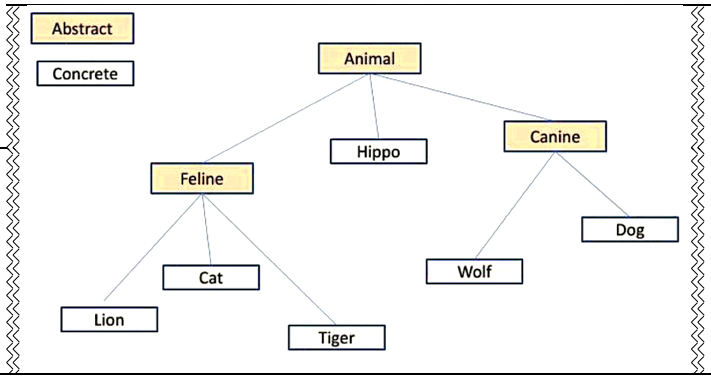
Which is unknown

- It deals with hiding the details and showing the essentials things to the user.
- You can achieve the abstraction in two ways:

I. ABSTRACT CLASSES

CLASSES

-we **can not** create new Animal
-we **can** create new Dog

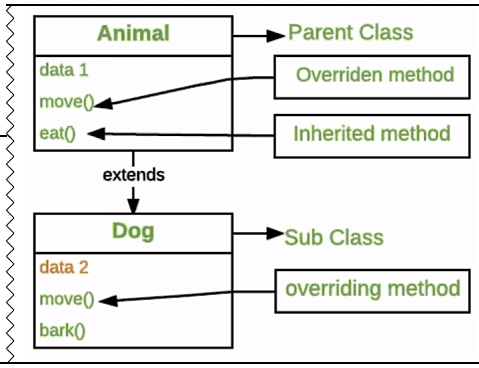



```
abstract class Canine extends Animal {

    //Code
}
```

- **Abstract classes** can not be instantiated
- An abstract class has virtually no use, no value, no purpose in life, unless it is **extended**
- With an abstract class, the guys doing the work at runtime are **instances of a subclass** of your abstract class.

ABSTRACT METHODS

-An abstract class means the class must be **extended**;
-An abstract class method means the method must be **overridden.**




```
public abstract void eat();
```

An abstract method has **NO** body!
-You already decided there is not any code that would make sense in the abstract, you won't put in a method body. So no curly braces, just end the declaration with a semicolon.
-If you declare an abstract method, you **MUST** mark the class abstract as well. You can not have an abstract method in a non-abstract class.

Summary;

- An abstract class is the one whose instances can not be created.
- Any class that has at least one abstract method has to be compulsory declared as an abstract class.
- Abstract class can contain both abstract and non-abstract methods
- An abstract method has no body
- Child class needs to override the all abstract methods.

II. INTERFACE

Interface in Java is a blueprint of a class or you can say it is a collection of abstract methods. In an interface, each method is public and abstract.

Interface also helps to achieve multiple inheritances in Java.

Example: We create Iphone and developer put "cook" or "dance" option on it and if it is not compulsory to create an iphone, developer put them implementing part and they are interface.

```
//To DEFINE an interface:
    public interface Pet{...}

//To IMPLEMENT an interface
    public class Dog extends Canine implements Pet{...}
```

Dog has access to Canine, Pet is interface

```
//All interface methods are abstract,so they MUST end in semicolon.
//They have no BODY
public interface Pet{

    public abstract void beFriendly();
    public abstract void Play();

}


public class Dog extends Canine implements Pet{

    public void beFriendly(){...} //You said you are a Pet, you must implement the Pet methods.It is your contract.

    public void play(){...} //You said you are a Pet, you must implement the Pet methods.It is your contract.

    public void roam(){...} //These are just normal overriding methods

    public void eat(){...} //These are just normal overriding
}
```

4- Polymorphism

Polymorphism means taking many forms. It is the ability of a variable, function, or object to take on multiple forms. Polymorphism in Java is 2 types:

I. Run time polymorphism (Dynamic Binding)

Method overriding is an example of run time polymorphism.

Rules for Java Method Overriding

- ✓ Method must have **same name** as in the parent class
- ✓ Method must have **same parameter** as in the parent class
- ✓ There must be IS-A relationship (**inheritance**)

```
class over{

    public void travel(){

        System.out.println("This method specifies the different forms of travel");
    }
}

public class override extends over{

    //Method is overriden here
    public void travel(){
        System.out.println("This method specifies car travel");
    }

    public static void main(String[] args){
        override r=new override();
        r.travel();
    }
}
```

II. Compile time polymorphism (Static Binding)

Method overloading is an example of compile time polymorphism. Method overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Unlike method overriding, arguments can differ in:

1. Number of parameters passed to a method
2. Data type of parameters
3. Sequence of data types when passed to a method

```
class Adder{

    static int add(int a , int b){
        return a+b;
    }

    static double add(double a, double b){
        return a+b;
    }
}


public static void main(String args[]){

    System.out.println(Adder.add(11,11));
    System.out.println(Adder.add(12.3,12.6));
}
```

	Overloading	**Overriding**
Definition	Methods having same name but each must have different number of parameters or parameters having different types & order	Sub class have method with same name and exactly the same number and type of parameters and same return type as super class method
Meaning	More than one method shares the same name in the class but having different signature	Method of base class is re- defined in the derived class having same signature
Behavior	To add/extend more to method's behavior	To change existing behavior of method
Polymorphism	Compile Time	Run Time
Inheritance	Not Required	Always Required
Method Signature	Must have different signature	Must have same signature

PROTOTYPE

1. ***CALLING FUNCTION***

<table>
<tr><td>Method #1</td><td>function foo(){
  console.log("Hello");
}
foo();</td><td>//Hello</td></tr>
<tr><td>Method #2</td><td>var obj = { "prop": " This is the object"};
obj.foo = function(){
  console.log("Hello");
  console.log(this);
};
obj.foo();</td><td>//Hello
//{prop: " This is the object ", foo: ƒ}
//this refers to the object whose property is the function that I am calling</td></tr>
<tr><td>Method #3</td><td>function foo(){
  //this={};
  console.log("Hello");
  console.log(this);
  //return this;
}
new foo(); //Hello //foo{};</td><td>--> new automatically create it
//Hello
//foo{}
--> new automatically create it</td></tr>
<tr><td rowspan="2">Method #4</td><td>function Bicycle(cadence, speed, gear, tirePressure){
  this.cadence=cadence;
  this.speed=speed;
  this.gear=gear;
  this.tirePressure=tirePressure;
  this.inflateTires=function (){
  this.tirePressure += 3;
  }
}
var bicycle1 = new Bicycle(50,20,4,25);
bicycle1.inflateTires();</td><td>//constructor mode

//28</td></tr>
<tr><td>function Mechanic(name){
  this.name=name;
}

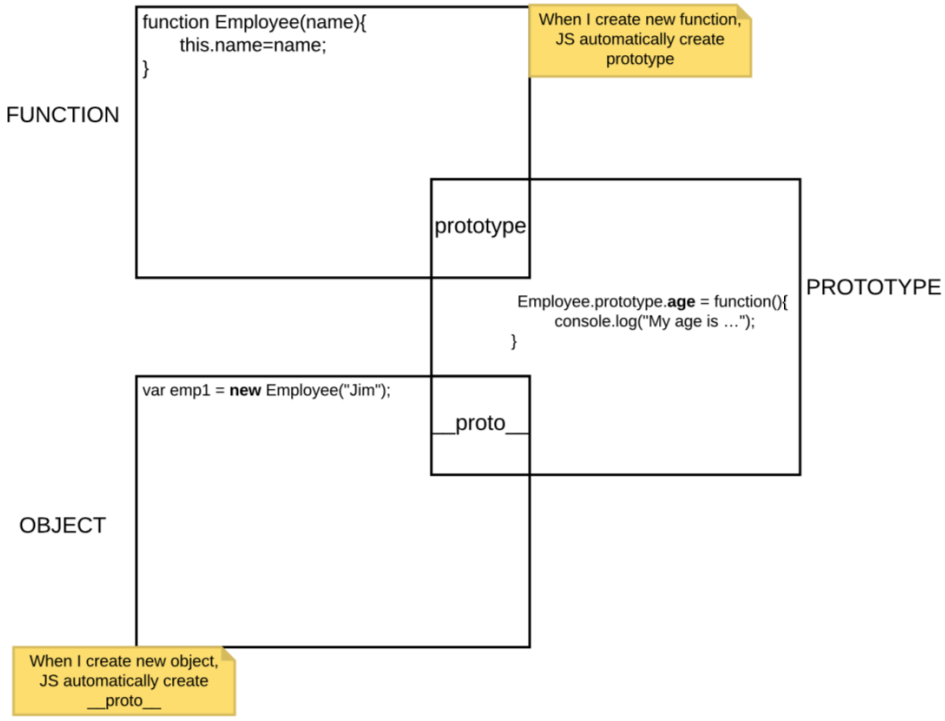
var mike= new Mechanic("Mike");
mike.inflateTires = bicycle1.inflateTires;
mike.inflateTires();

mike.inflateTires.call(bicycle1);</td><td>EXTRA

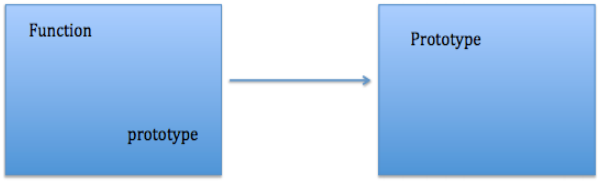
// undefined so we need 4th method

//undefined --> before this call bicycle1 tirePressure is 25; After calling this function bicycle1 tire pressure will be 28</td></tr>
</table>

2. ***PROTOTYPE***



When we create a function (foo) JS create 2 objects 1- foo object 2- prototype object	function foo(){ };	
Access to foo object→	foo;	// function foo(){}
Access to prototype→	foo.prototype;	// {constructor function}

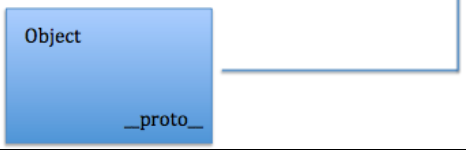


When we create a **new** object JS creates an object **__proto__**	var newFooObj= **new** foo();

foo function→

← foo.prototype (father)

newFooObj→ (child)



I add a property on foo.prototype	foo.prototype.**test**= "I am a tester";

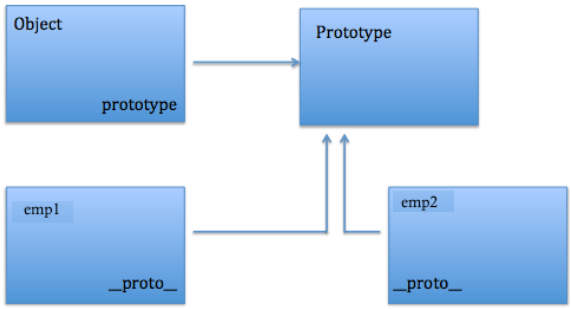
How can I access to **.test** property		
1st way	foo.prototype.test	//I am a tester
2nd way	newFooObj.**__proto__**.test	//I am a tester

So;	foo.prototype.test === newFooObj.**__proto__**.test	//true

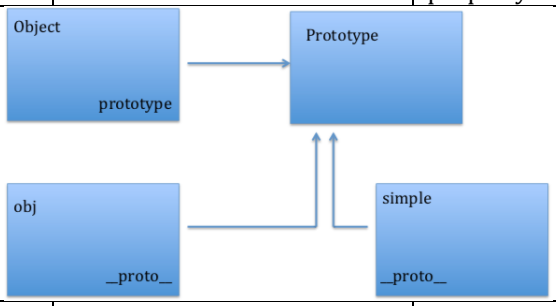
ANOTHER ONE

Create a function	function Employee(name){ this.name=name; }	
Creating new objects to our template	var emp1 = new Employee("Jim"); var emp2 = new Employee("Pan");	JS creates **__proto__** property for each **new** one

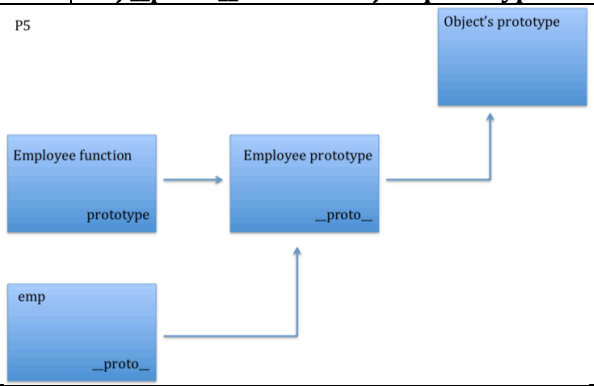
Adding a property on prototype	Employee.prototype.playPranks = function(){ console.log("Prank Played"); }



How can I access to new property **newPranks** by new objects	emp1.__proto__. playPranks (); // Prank Played emp2.__proto__. playPranks (); // Prank Played	
or →→→→→→	emp1.age();	JS checks emp1 has this property or not →if the object does not have this property; JS always ask the __proto__, "Hey do you have this property?" and if it is yes, it executes



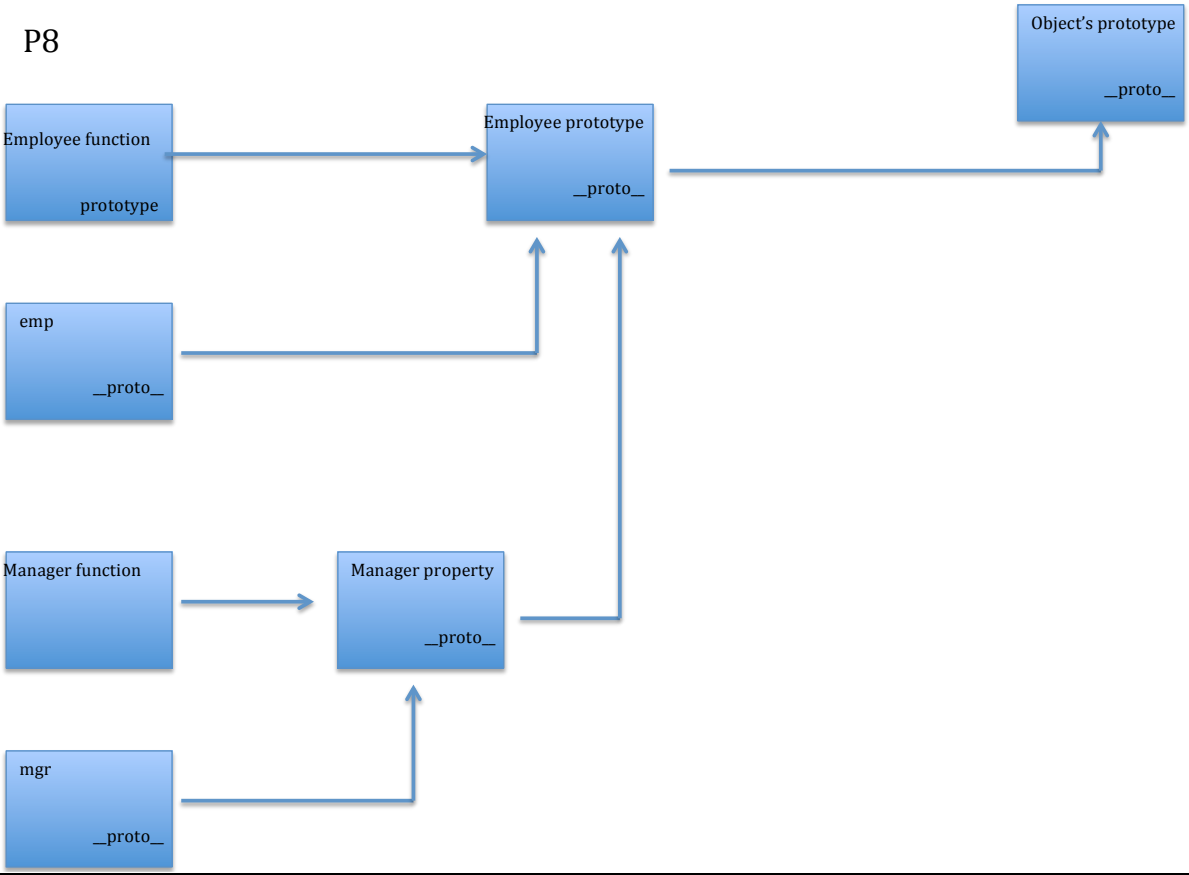
create function (Object)	function Object(){}	JS create a prototype for Object
create object	var simple = {};	JS create __proto__
create a new object (obj)	var obj=new Object();	JS create __proto__ for obj
→→→	**simple.__proto__ === obj.__proto__ //true** **obj.__proto__ === Object.prototype //true**	



We create a function (Employee)	function Employee(){}	JS create a **prototype** for **Employee**
We create a new object	var emp=new Employee();	JS create **__proto__** for **emp**
→→→	console.log(emp.__proto__.__proto__ === Object.prototype) //true	
→→→	console.log(emp.__proto__.__proto__.__proto__); //null	

MULTILEVEL PROTOTYPE

P8



create Employee function	function Employee(name){ this.name = name; }	
add a property our prototype	Employee.prototype.**getName** = function(){ return this.name; }	
create a new object	var emp1=new Employee("Mike");	//instance
	console.log(emp1.getName());	//Mike

create Manager function	function Manager(name, dept){ this.name=name; this.dept=dept; }	
add a property our prototype	Manager.prototype.**getDept**=function(){ return this.dept; }	
create a new object	var mgr = new Manager("Michael", "Sales");	
	console.log(mgr.getDept());	//Sales
→→→	mgr.__proto__.__proto__ = Employee.prototype //to reach getName	
	console.log(mgr.getName()); // Michael	
	Employee.prototype.getSalary = function () { return 100; }	
	mgr.getSalary(); //100	

ES6

1. ***Variable Declarations with let and const***

let **const**	1- We don't use **var** anymore 2- Variables declared with **var** in ES5 is function **scoped**, and the variables declared with **let** and **const** in ES6 are **block-scoped**.

ES5	ES6
`var name5 = "Mike Smith";` `var age5 = 23;` `name5 = "Mike Miller";` `console.log(name5); //Mike Miller`	`const name6 = "Mike Smith";` `let age6 = 23;` `name6 = "Mike Miller"` `console.log(name6) //Error: Assignment to constant variable; because we can not change`

```
function driverLicence5(passedTest){
    if (passedTest){
        var firstName="Mike";
        var yearofBirth= 1970;
        console.log(firstName + ", born in " + yearofBirth + " is now officially allowed to drive")
    }
     //console.log(firstName + ", born in " + yearofBirth + " is now officially allowed to drive") //It works because var is coming ES5 and it is scoped.
}
//console.log(firstName + ", born in " + yearofBirth + " is now officially allowed to drive") //It doesn't work. var is outside of the function

driverLicence5(true); //Mike, born in 1970 is now officially allowed to drive
```

```
function driverLicence5(passedTest){
    if (passedTest){
        let firstName2="Mike";
        const yearofBirth2= 1970;
        console.log(firstName2 + ", born in " + yearofBirth2 + " is now officially allowed to drive") //It works because of block-scope
    }
      //console.log(firstName2 + ", born in " + yearofBirth2 + " is now officially allowed to drive") // It does not work because my variables is out of scope. My console.log is out of the if{}.
}
driverLicence5(true);
```

```
var i=23;
for (var i=0;i<5;i++){
    console.log(i);   // 0, 1, 2, 3, 4
}
console.log(i); // 5
```

```
let i=23;
for (let i=0;i<5;i++){  // this "let i" is different from the first "let i"
    console.log(i);   // 0, 1, 2, 3, 4
}
console.log(i); // 23 -->let is block-scope so not get info inside {}
```

```
let i=23;
let sum=0;
for (i=0;i<5;i++){  // this "i" is same with the first "let i"
    console.log(i);   // 0, 1, 2, 3, 4
    sum +=i;
}
console.log(i); // 5
console.log(sum); //10
```

2. *Template Strings*

` ` **${ }**	1-` ` instead of " " 2- We don't need to use concatenation

ES5	ES6
`var a = 10;` `var b = 10;` `console.log("JS first appeared " + (a+b) + " years ago")`	`var a = 10;` `var b = 10;` `console.log(`JS first appeared ${a+b} years ago.`); //${}`

```
let firstName = "Mike";
let lastName = "Smith";
const yearOfBirth = 1970;
function calcAge(year){
    return 2018 - year;
}
console.log(`This is ${firstName} ${lastName}. He was born in ${yearOfBirth}. Today, he is ${calcAge(yearOfBirth)} years old`)
```

.startsWith( ) **.endsWith( )** **.includes( )**	**result is Boolean (true / false)**
.repeat( )	repeat (how many times)

```
let firstName = "Mike";
let lastName = "Smith";

const n = `${firstName} ${lastName}`;
console.log(n.startsWith("M"));  //true
console.log(n.endsWith("th"));  //true
console.log(n.includes("ike"));  //true
console.log(firstName.repeat(5));
 //MikeMikeMikeMikeMike

console.log(`${firstName} `.repeat(5));
 //Mike Mike Mike Mike Mike
```

3. *Arrow Functions*

let x = a => a;	let function name = parameter => return

ES5	**ES6**
`var x = function(a){ return a; } console.log(x(2)); //2`	`let x = a => a;`
If we don't have any parameter	
	`let x = () => {console.log(`Hello`)}; x();`
If we have multiple parameters	
`var z = function(a,b,c){ return a+b+c; } z(1,2,4); //7`	`let z = (a,b,c) => a+b+c z(1,2,4); //7`

. map( )	**create new array** with the result of calling a provided function on every element in the calling array
pass a function to map→	`const array1 = [1, 4, 9, 16]; const map1 = array1.map(x=>x*2); console.log(map1); //New Array [2, 8, 18, 32] console.log(array1); // [1, 4, 9, 16]`
Task→	`const years = [1990, 1965, 1982, 1957]; let ages6 = years.map(el => 2018 - el); console.log(ages6); // New Array [28, 53, 36, 61]`
Task→	`var materials = [ "Hydrogen", "Helium", "Lithium", "Beryllium" ] //Expected Outcome : [8,6,7,9]; let y = materials.map(x=> x.length); console.log(x);`

Arrow functions **CANNOT** be used as a **constructor** and will throw an error when used with new.→	`var foo = () =>{} var newf = new foo();`
Arrow functions do NOT have a prototype property→	`var foo = () =>{} console.log(foo.prototype); // undefined`

Destructuring	It is a JS expressions that makes it possible to unpack values from array, or properties from objects, into distinct variables.

ES5	ES6
`var john = ["jonh", 26];` `var name = john[0];` `var age = john[1];`	`let [name,age] = ["john", 26];` `name;  //john`

```
var obj = {
    firstName : "Mike",
    lastName : "Smith"
}
console.log(obj.firstName);
console.log(obj.lastName);
```

```
const obj = {
    firstName : "Mike",
    lastName : "Smith"
}
const {firstName, lastName, age} = obj;
console.log(firstName);     //Mike
console.log(age);           //undefined
```

```
function calculateRetirement(year){
    const age = new Date().getFullYear()
    - year;
    return [age,65-age];
}

const [age2,retirement] =
calculateRetirement(1990);

console.log(age2);          //28
console.log(retirement);    //37
```

4. ***Arrays***

Array.from( )	It creates a new, shallow-copied Array instance from an array-like or iterable object.

ES5

ES6

```
var x =Array.from("JavaScript");
console.log(x);
//["J","a","v","a","S","c","r","i","p","t")]
```

.findIndex( )	returns the INDEX of the first element in the array that satisfies the provided texting f.
.find( )	returns the VALUE of the first element in the array that satisfies the provided texting f.

ES5

```
var ages = [12, 17, 8, 21, 14, 11];
var full = ages.map(function(cur) {
    return cur >= 18;
});
console.log(full);
//[false, false, false, true, false, false]
console.log(full.indexOf(true)); //3
console.log(ages[full.indexOf(true)]); //21
```

ES6

```
let ages = [12, 17, 8, 21, 14, 11];
let full = ages.map(function(cur) {
    return cur >= 18;
});
console.log(ages.findIndex(cur => cur >= 18));//3
console.log(ages.find(cur => cur >= 18));    //21
```

for..of	It creates a loop iterating over iterable objects. ES5; we can limit with iteration, but in ES6; we CANNOT limit

ES6

Array→

```
let myArray = [10,20,30];
for (let x of myArray){
    x +=1;
    console.log(x);   //11, 21, 31
}
```

Multidimensional Array→

```
let myMulti = [[1,2,3],[4,5,6]];
for(let x of myMulti){
    for(let y of x){
        console.log(y); //1,2,3,4,5,6
    }
}
```

Object→

```
let myObject = {
    x:1,
    y:2,
    z:3
}
for (let a in myObject){
    console.log(a);          //x,y,z
    console.log(myObject[a]);   //1,2,3
}
```

for … in→

```
let list = [4,5,6];
for (let i in list){  //index numbers
    console.log(i);   //0,1,2
}
```

for … of→

```
let list = [4,5,6];
for (let i of list){  //values
    console.log(i);   //4,5,6
}
```

5. ***Spread Operator***

...	**Spread syntax** allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

ES5	ES6
	`let mid = [3,4];` `let arr = [1,2,mid,5,6];` `console.log(arr);  // [1, 2, [3, 4], 5, 6]` `let arr2 = [1,2,...mid,5,6]  //spread operator` `console.log(arr2)  //[1, 2, 3, 4, 5, 6]`

```
function addFourAges(a, b, c, d) {
    return a + b + c + d;
}
var sum1 = addFourAges(10, 20, 30, 40);
console.log(sum1);          //100
```

if I have those ages in an array, how can I pass it to function? →

```
function addFourAges(a, b, c, d) {
    return a + b + c + d;
}
let ages = [10,20,30,40];
let sum = addFourAges(...ages);
//let sum = (a,b,c,d) => a+b+c+d;
//console.log(sum(...ages)); //100
console.log(sum);  // 100
```

...	**Rest Parameters** the big difference between spread and rest operator is the place in which we use each of them spread operator is used in the function call rest operator is used in the function declaration to exact an arbitrary number of parameters.

```
function sumAll(...args){
    let sum = 0;
    for(let arg of args){
        sum+=arg;
    }
    return sum;
}
console.log(sumAll(1));
console.log(sumAll(1,2));
console.log(sumAll(1,2,3));
```

How can we write this function with arrow functions;

```
let multiply = (multiplier, ...theArgs)
=>theArgs.map(element=>multiplier*element);
```

```
function multiply(multiplier, ...theArgs){
    return theArgs.map(function(element){
        return multiplier * element;
    })
}
let arr = multiply(2,1,2,3);
console.log(arr);       //[2, 4, 6]
```

6. ***Maps***

new Map ( )	The map object holds **key-value pairs**. Any value (both objects and primitive values) may be used as a key or a value
.set( )	specify key and value
.get( ) **.size( )** **.entries( )**	

	ES6
	`myMap.set("keyString","value with a string");` `myMap.set("keyObj", "value with keyObj");` `myMap.set("keyFunc", "value with keyFunc");` `console.log(myMap.size);  //3` `console.log(myMap.get("keyString"));` `//value with a string`

```
let question = new Map();
question.set('question', 'What is the official
name of the latest major JS version?');
question.set(1, 'ES5');
question.set(2, 'ES6');
question.set(3, 'ES2015');
question.set('correct', 3);
question.set(true, 'Correct answer is D');
question.set(false, 'Wrong, try again!');

console.log(question.get('question')); //What
is the official name of the latest major JS
version?
console.log(question.size); //7

for (let [key, value] of question.entries()) {
    console.log(`Answer ${key}: ${value}`);
}
```

7. Classes and Subclasses

	- Introduced in ECMAScript 2015, are primarily syntactical sugar over JS's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JS. - in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.
instanceof	`class Person {` `  //details` `}` `let person = new Person();` `console.log(person instanceof Person);//true`
constructor static	When to use static methods? 1-If you are writing utility classes and they are not supposed to be changed. 2-If there is some code that can easily be shared by all instance methods, extract that code into a static code 3-If you are sure that the definition of the method will never be changed or overridden. Static methods can not be overridden.
Instance Variable	One copy per object. Every object has its own instance variable.
Static Variable	One copy per class. The static keyword defines a static method for a class. Static methods are called without instantiating their class and cannot be called through a class instance.

ES5

```
var Person5 = function(name,
yearOfBirth, job) {
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}
Person5.prototype.calculateAge =
function() {
    var age = new Date().getFullYear()
    - this.yearOfBirth;
    console.log(age);
}
var john5 = new Person5('Mike', 1980,
'tester');
mike5.calculateAge();   //38
```

ES6

```
class Person6 {
    constructor (name,yearOfBirth,job){
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.job = job;
    }

    calculateAge(){
        var age=new Date().getFullYear()-
        this.yearOfBirth;
        console.log(age);
    }

    static greetings(){
        console.log('Hello');
    }
}
let mike6=new Person6('Mike',1980,'tester');
mike6.calculateAge(); //38
Person6.greetings();  //Hello
```

constructor

prototype

static

Terms for interview→

```
//calling a constructor
let mike6 = new Person6();
//passing arguments into constructor
let mike6 = new Person6("Mike",1980,"tester");
//this keyword to refer to the instance being created
console.log("Mike's job is " +mike6.job)
```

extends **super**	**INHERITANCE** keywords - If there is a constructor present in subclass, it needs to first call **super( )** before using **"this"**

	ES6
Inheriting Properties→	`class Person{ //super class - parent class` `constructor(name,age,weight){` `this.name=name;` `this.age=age;` `this.weight=weight;` `}` `displayName(){` `console.log(this.name);` `}` `displayAge(){` `console.log(this.age);` `}` `displayWeight(){` `console.log(this.weight);` `}` `}` `class Programmer extends Person{` `constructor(name,age,weight,language){` `super(name,age,weight); //call the super class constructor` `this.language=language;` `}` `displayLanguage(){` `console.log(this.language);` `}` `}` `let mike = new Person("mike",25,200);` `console.log(mike instanceof Programmer); //true` `console.log(mike instanceof Person); //true` `console.log(mike instanceof Object); //true` `mike.displayName();` `mike.displayAge();` `mike.displayWeight();` `console.log('.....')` `let john = new Programmer("john",30,180,"JS");` `john.displayName();` `john.displayAge();` `john.displayWeight();` `john.displayLanguage();`
Inheriting Methods→	`class Vehicle{` `start(){` `console.log("starting the vehicle");` `}` `}` `class Car extends Vehicle{` `start(){` `//super.start();` `console.log("starting the car")` `}` `}` `let honda = new Car();` `honda.start(); //starting the car`

get **set**	- Getters and setters are functions or methods used to get and set the values of variables. - The idea of getter and setter is always mentioned in conjunction with encapsulation. **Encapsulation** is done to hide data, to make it inaccessible from other code, except through the getters and setters. This way we do not end up accidentally overwriting important data with some other code in the program.

	ES6
	`` class GetThings{ constructor(size){ this.length=size; } get Length(){ return this.length; } set Length(value){ this.length=value; console.log("the value has been set"); } } var thing = new GetThings(9); console.log(thing.Length); //9 ->no braces() thing.Length=10; console.log(thing.Length);//Value has been set //10 ```
	`` class EncapTest{ constructor(name,idNum,age){ this._name = name; this._idNum = idNum; this._age = age; } get Age(){ return this.age; } get Name(){ return this.name; } get IdNum(){ return this.idNum; } set Age(newAge){ this.age = newAge; } set Name(newName){ this.name = newName; } set IdNum(newId){ this.idNum = newId; } } var encap = new EncapTest(); encap.Name="Mike"; encap.Age=20; encap.IdNum="MS30"; console.log(`Name : ${encap.Name} Age : ${encap.Age} ID : ${encap.IdNum}`); //Name : Mike Age : 20 ID : MS30 var encap2 = new EncapTest("Mike","MS30",30); console.log(`Name : ${encap2.Name} Age : ${encap2.Age} ID : ${encap2.IdNum}`); //Name : undefined Age : undefined ID : undefined ```

Callback Function, Closures, call ( ), apply ( ), bind ( )

Callback Functions	Functions are **First Class Objects**. Functions have the ability to: 1-Be assigned to variables 2-Have other functions in them 3-Return other functions to be called later ➢A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action ➢A callback function is function that you intend to execute at a future date when some action has been completed or some event triggers them to run.

simply→

```
let x  = function(){
    console.log("I am called from inside a function");
}
let y = function(callback){
    console.log("do smthg");
    callback();
}
y(x);   //do smthg
        //I am called from inside a function
```

if this function calc is a part of some library, user supposed to use this library and pass the number and calculation type and should get the result. In order for all the possible calculation type, this library has to implement of all of it→

```
let add = function(a,b){
    return a+b;
}
let multiply = function(a,b){
    return a*b;
}
let calc = function(num1,num2,callback){
    return callback(num1,num2);
}
console.log(calc(2,3,add));
console.log(calc(2,3,multiply));
```

I can write function directly as an argument→

```
//anonymous function - there is no function name
(if I want to use it only once)
console.log(calc(2,3,function(a,b){
    return a-b;
}));
```

when you have to use something only once, you do not want to define it and pass it. You can just directly write into function argument modify→

```
let calc = function(num1,num2,callback){
    if(typeof callback == "function"){
        return callback(num1,num2);
    }
}
```

TASK

write a function that computes the total of all the scores together for East and also account so how many students are in our in East

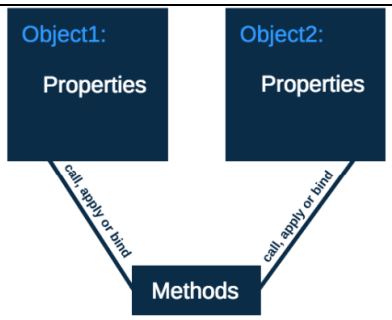
```
let students = [
    {name:"Mary",score:90,school:"East"},
    {name:"James",score:100,school:"east"},
    {name:"Steve",score:40,school:"East"},
    {name:"Gabe",score:90,school:"West"},
    {name:"Racheal",score:85,school:"East"},
    {name:"Mike",score:95,school:"West"},
    {name:"Smith",score:75,school:"East"},
];
let processStudents = function(data,callback){
    for(let i=0;i<data.length;i++){
        if(data[i].school.toLowerCase()=="east"){
            if(typeof callback == "function"){
                callback(data[i]);
            }
        }
    }
}
processStudents(students,function(x){
    if(x.score>60){
        console.log(x.name + " passed.");
    }
});
```

```
let determineTotal = function(){
    let total = 0;
    let count = 0;
    processStudents(students,function(x){
        total = total + x.score;
        count++;
    });
    console.log("Total score: " + total + " - Total
    Count: " + count);
}
determineTotal();
        //Total score: 390 – Total Count: 5
```

Closures	* Key to understanding closures is understanding how FUNCTIONS WITHIN FUNCTIONS work An inner function has always access to the variables and parameters of its outer function, even after the outer function has returned. 

Function Returning a Value	`function calculateRectangleArea(length,width){` `  return length * width;` `}` `var roomArea = calculateRectangleArea(10,10);` `console.log(roomArea);  //100`
Function Returning a Function	`function youSayGoodBye(){` `  console.log("Good Bye");` `  function andISayHello(){` `  console.log("Hello");` `  }` `  return andISayHello;` `}` `var something = youSayGoodBye();` `console.log(something); //Good Bye` `something();  //Good Bye / Hello`
Closures get involved when the returned inner function is not SELF-CONTAINED.	`function stopWatch(){` `  var startTime = Date.now();` `  function getDelay(){` `  var elapsedTime=Date.now()-startTime;` `  console.log(elapsedTime);` `  }` `  return getDelay;` `}` `var timer = stopWatch();` `  function getDelay(){` `  var elapsedTime=Date.now()-startTime;` `  console.log(elapsedTime);` `  }`
JavaScript runtime keeps track of all of your variables, memory usage, references, and so on. When it detects that an inner function relies on variables stored by an outer function, it ensures those variables are available even if the outer function goes away.	`function retirement(retirementAge){` `  var a =" years left until retirement.";` `  return function(yearOfBirth){` `  var age = 2018 - yearOfBirth;` `  console.log((retirementAge-age)+a);` `  }` `}` `var retirementUS=retirement(66);` `retirementUS(1983); //31 years left until retirement.` `retirement(66)(1983);//31 years left until retirement.`

.call( )	The call( ) method calls a function with a given this value and arguments provided individually.
.apply( )	The apply( ) method calls a function with a given this value, and arguments provided as an array
.bind( )	bind( ) returns a new function, allowing you to pass in a this array and any number of arguments.



.call( )

```
//I have object
var obj = {
    num:2
}
//I have a method outside of the object
var addToThis = function(a){
    return this.num + a;
}
//I can call the object property and use it in the
function
var x = addToThis.call(obj,3);
console.log(x);

//If we have more than one argument
var obj3 = {
    num1: 10,
    num2: 5
}
var addToThis = function(a,b){
    return this.num1 + this.num2 + a*b;
}
var x = addToThis.call(obj3,3);
console.log(x);
```

.apply( )

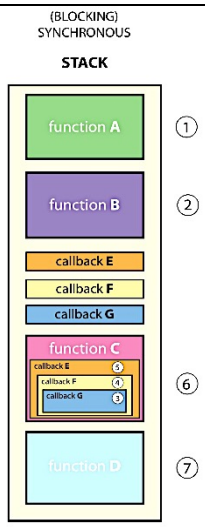
```
//I have an object
var obj4= {
    num:10
}
//I have an array
var arr = [1,2,3];
//I have a method
var addToThis = function(a,b,c){
    return this.num + a + b + c;
}
var x = addToThis.apply(obj4,arr)
console.log(x);
```

.bind( )

```
var obj = {
    num:2
}
var addToThis = function(a,b,c){
    return this.num+a+b+c;
}
var arr = [1,2,3];
console.log(addToThis.bind(obj,arr));

var b = addToThis.bind(obj);
console.log(b(1,2,3)); //8
console.log(b(...arr)); //8
```

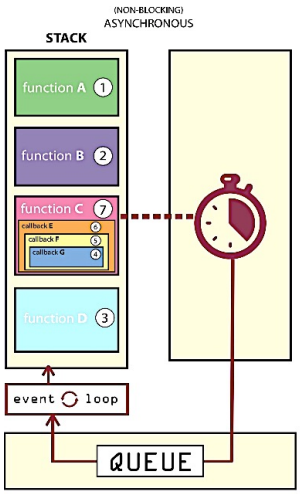
Synchronous – Asynchronous / PROMISE

SYNCHRONOUS
JavaScript is a synchronous, blocking, single-threaded language. That just means that only one operation can be in progress at a time.




```
function first(){
    console.log("first one");
}
function second(){
    console.log("second one");
}
function third(){
    console.log("third one");
}
first();
second();
third();
```

ASYNCHRONOUS
The earliest and most straightforward solution to being stuck in the synchronous world was asynchronous callbacks (think **setTimeout( )**).




```
function first(){
    setTimeout(function(){
    console.log("first one");
    },1000)
}
function second(){

    console.log("second one");
}
function third(){

    console.log("third one");
}
first();
second();
third();     //second one
             //third one
             //first one, after 1 sc
```

callback function

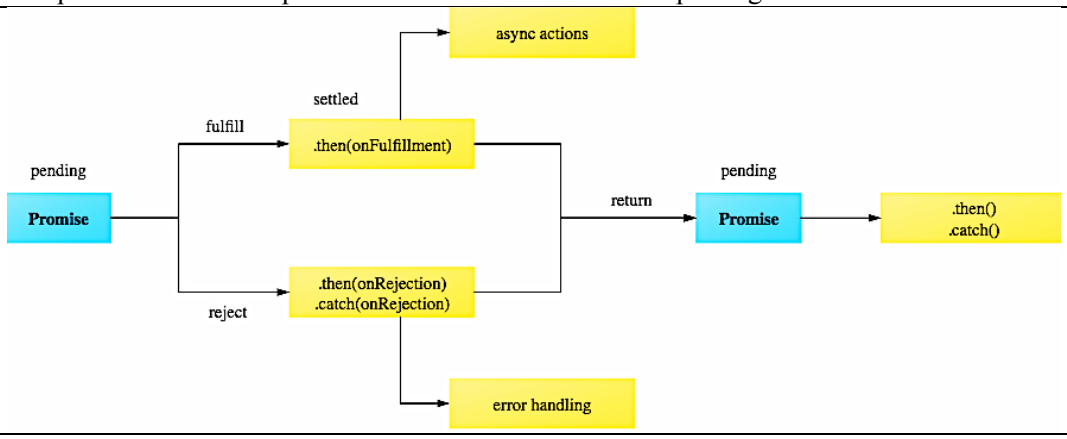
```
function first(callback){
    setTimeout(function(){
        console.log("first one");
        callback(third); //second(third)
    },1000)
}
function second(callback){
    console.log("second one");
    callback(); //third()
}
function third(){
    console.log("third one");
}
first(second);
```

Callback Hell / Pyramid of Doom

```
function prepareTea(){
    setTimeout(function(){
        console.log("1.Find Vessel")
        setTimeout(function(){
            console.log("2.Put water, sugar and tea");
            setTimeout(function(){
                console.log("3.Bring the mixture to a boil");
                setTimeout(function(){
                    console.log("4.Filter the tea");
                    setTimeout(function(){
                        console.log("5.Serve the tea");
                    },2000)
                },2000)
            })
        },2000)
    },2000)
}
prepareTea();
```

Promises

➢Object that keeps track about whether a certain event has happened already or not;
➢Determines what happens after the event has happened
➢Implements the concept of a future value that we are expecting



Promise Constructor

The promise object represents the eventual completion(or failure) of an asynchronous operation, and its resulting value.

```
new Promise(
    /* executor */
    function(resolve, reject){
        ...
    });
```

Executer : A function that is passed wit the arguments **resolve** and **reject**. The executer Function is executed immediately by the Promise implementation, passing resolve and reject functions. The **resolve** and **reject** functions, when called, resolve or reject the promise, respectively. The executer normally initiates some asynchronous work, and then once that completes, either calls the **resolve** function to resolve the promise or else rejects it if an error occurred. If an error is thrown in the executer function, the promise is rejected. The return value of the executer is ignored.

Create a Promise →

A promise object is created using the new keyword and its constructor. This constructor takes as its argument a function, called the 'executer function' . This function takes 2 parameters : resolve and reject.

```
let promiseToCleanTheRoom = new Promise((resolve,reject)=>{
    //do stg asynchronous that eventually calls either
    let isClean = true;
    if (isClean){
        resolve("Clean");  //fullfilled
    }else{
        reject("Not Clean"); //rejected
    }
})
```

.then( ) method→

returns a Promise. It takes up to two arguments: callback f for the success and failure cases of the Promise.

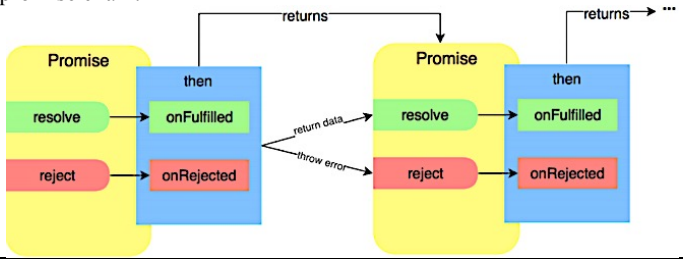
.catch( ) method→

returns a Promise and deals with rejected cases only.

```
promiseToCleanTheRoom.then(function(fromResolve){
    console.log("The room is " + fromResolve);
}).catch(function(fromReject){
    console.log("The room is " + fromReject);
}) //The room is Clean -> because isClean=true
```

If we have more than one dependencies and a function parameter

```
let cleanRoom = function(){
    return new Promise(function(resolve,reject){
        resolve(" Cleaned the room")
    })
}
let removeGarbage = function(msg){
    return new Promise(function(resolve,reject){
        resolve(msg + " Removed garbage");
    })
}
let winIcecream = function(msg){
    return new Promise(function(resolve,reject){
        resolve(msg + " Won Icecream");
    })
}
cleanRoom().then(function(result){
    return removeGarbage(result);
}).then(function(result){
    return winIcecream(result);
}).then(function(result){
    console.log("finished "+result);
}) //finished Cleaned the room Removed garbage Won Icecream
```

Benefits of Promises	➢Callbacks will never be called before the completion of the current run of the JavaScript event loop. ➢Callbacks added with .then() even after the success or failure of the asynchronous operation, will be called. ➢Multiple callbacks may be added by calling .then() several times, to be executed independently in insertion order. ➢Chaining
Chaining	➢A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from previous step. We accomplish this by creating a promise chain. 

async / await	➢The newest way to write asynchronous code in JavaScript ➢It is non blocking (just like promises and callbacks) ➢Async/Await was created to simplify the process of working with and writing chained promises ➢Async functions return a Promise. If the function throws error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

	`` const async = () => { return new Promise((resolve,reject)=>{ resolve("Everything is good"); }) }; async() .then((data)=>{ console.log(data);//Everything is good return 1; }) .then((data)=>{ //1 is going to data console.log(data); //1 return 2; }) .then((data)=>{ //2 is going to data console.log(data); //2 }) ```
Create promise chain without async / await	``` function doubleAfter2Seconds(x){ return new Promise((resolve,reject)=>{ setTimeout(()=>{ resolve(x*2); },2000) }) } function addPromise(x){ return new Promise(resolve=>{ doubleAfter2Seconds(10).then((a)=>{ doubleAfter2Seconds(20).then((b)=>{ doubleAfter2Seconds(30).then((c)=>{ resolve(x+a+b+c) }) }) }) }) } addPromise(10).then((sum)=>{ console.log(sum) //130 }) ```
Create promise chain with async / await	``` function doubleAfter2Seconds(x){ return new Promise((resolve,reject)=>{ setTimeout(()=>{ resolve(x*2); },2000) }) } async function addAsync(x){ const a = await doubleAfter2Seconds(10); const b = await doubleAfter2Seconds(20); const c = await doubleAfter2Seconds(30); return x+a+b+c } addAsync(10).then((sum)=>{ console.log(sum); //130 }) ```

Error Handling

Exception is an **unwanted** or **unexpected event**, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

For example referencing a variable or a method that is not defined can cause an exception.
Key Words:

- try
- catch
- throw
- finally

try … catch block

- The JS statements that can possibly cause exceptions should be wrapped inside a **try** block.
- When a specific line in the **try** block causes an exception, the control is immediately transferred to the **catch** block skipping the rest of the code in the try block.

```
try
{
  //do anything
}
catch(error)
{
  //don't forget errors
}
```

```
try{
    console.log("Hello");
    console.log(sayHello());
    console.log("This is an error");
}

catch(e){
    console.log("I see your error");
    //console.log("Description: " + e.description);
    console.log("Message: " + e.message);
    console.log("Stack trace: " + e.stack);
}
```

```
Hello
Description: undefined
Message: sayHello is not defined
Stack trace: ReferenceError: sayHello is not defined
    at http://127.0.0.1:63542/errorHandli
```

finally block

- is guaranteed to execute irrespective of whether there is an exception or not. It is generally used to clean and free resources that the script was holding onto during the program execution. For example, if your script in the try block has opened a file for processing, and if there is an exception, the finally block can be used to close the file.

```
try{
    //block of code to try
}
catch(e){
    //block of code to handle errors
}

finally{
    //block of code to be executed
    regardless of the try/catch result
}
```

```
try{
    console.log("Hello");
    console.log(sayHello());
    console.log("This is an error");
}

catch(e){
    console.log("I see your error");
    console.log("Message: " + e.message);
    console.log("Stack trace: " + e.stack);
}

finally{
    console.log("It is guaranteed to execute")
}
```

```
Hello
I see your error
Message: sayHello is not defined
Stack trace: ReferenceError: sayHello is not defined
    at http://127.0.0.1:63542/err
it is guaranteed to execute
```

throw statement

- allows you to create a custom error
- the exception can be a JS String, a Number, a Boolean, or an Object.

```
throw "too big"; //Throw a text
throw 500;   //Throw a number
throw true;  //Throw a boolean
throw {toString: function(){return "I am an object"}}; //Throw an Object
```

```
function divide(){
    var num1=+(prompt("enter num1"));
    var num2=+(prompt("enter num2"));
    try{
        if (num2==0){
            throw{
                error: "Divide by zero error",
                message: "Number cannot be zero"
            }
        }else{
            console.log(`Result: ${num1/num2}`);
        }
    }
    catch(e){
        console.log(e.error);
        console.log(e.message);
    }
}
divide();
```

If the num2 =0;

It will print;
//Divide by zero error
//Number cannot be zero

TASK:

```
function f(){

    try{
        console.log(0);
        console.log(name2);
    }catch(e){
        console.log(1);
        return true;
        console.log(2);
    }finally{
        console.log(3);
        return false;
        console.log(4);
    }

    console.log(5);
}

console.log(f());
//0,1,3,false
f();
//0,1,3
```