- **var** vs **let :**
  var x means the variable is defined in block or function , whereas let defines it locally only.

- **const:**
  We can change constant values locally. Although if not redefined globally it retains same value everywhere else.

  e.g.
  const X = 12;
  if (true) {
  	const x = 123;
  	console.log(x); ← 123
  }
  console.log(x); ← 12

- **Data types:**
  typeof X ← gives data type of X ; parenthesis are optional.

  Boolean , Number , String , Symbol , undefined and objects exist (*Arrays are also objects*).

- **String Interpolation using template literals:**
  Use variable values inside strings like this -

  let x = 123;

  console.log(`Value of x is ${x}`) ← logs Value of x is 123.

  Template Literals can evaluate the value as well like - *`${20*10}`* ← means 200

  Tag Template Literals can modify outputs using functions as -

  Eg.
  function modify(...values) {
  	console.log(values);
  }

  Modify `${10}hello${20}from${30}here${40}` ← what it emits is an array having an array of all the strings (including null) from this string with other elements as 10 , 20 ,30 and 40.

  Ie. [[‘’,’hello’,’from’,’here’,’’],10,20,30,40]
  Mind the null strings on start and end they are caused by nothing before ${10} and nothing after ${40}

  That behaviour is because of **Rest Parameter.**
  Rest parameters hold Array of all values except those catched explicitly.
  e.g. func(str, ...values) ← will catch all strings array and then values will contain 10,20,30,40.
  Rest Parameters are not some sort of type. So we can’t do

let ...values

- **Playing with strings :**
    1. “ashish”.repeat(3) ← repeats ashish 3 times i.e. ashish ashish ashish
    2. “ashish”.startsWith(“as”) ← true
    3. “ashish”.endsWith(“sh”) ← true
    4. “ashish”.includes(“shis”) ← true
    5. ` string ` ← can be multiline without \ break.
- **Functions with default values :**
  Just like in Python -

  ```
  function myfunc( x=1 , y =2 ) {
          console.log(x+y);
  }
  ```

  myfunc(); ← prints 3

  myfunc(10,20) ← prints 30

  To print arguments array we can do -

  console.log(arguments);

  It outputs an **Associative Array** with 0 , 1 , 2 as keys.
- **Arrow Functions -**

  ```
  ( params ) => {
          // body
  }
  ```

- **Map, Reduce and Filter -**
  Given an iterable like an Array do -

  [1,2,3,4]**.reduce**((a,b) => a+b) ← take two values each time and return the sum of those till not even a single value exists i.e.
  1+2 ,3 ,4
  1+2+3,4
  1+2+3+4
  Outputs 10

  [1,2,3,4]**.map**((a) => 2*a); ← apply this function to all the values in the array. Outputs (2,4,6,8)

  [1,2,3,4]**.filter**(a => a%2==0); ← return the value a if the condition is satisfied. Outputs (2,4)

- **Objects -**
  Objects can have objects inside them as well.
  To get all members in an object do -

  Object.getOwnPropertyNames(object_name); ← outputs an array of all properties.

- **Destructuring Objects -**
  let { property_1, property_2 } = object

  object's property_1 gets assigned to property_1 variable and object's property_2 gets assigned to property_2 variable.

  Mind that property_1 and property_2 are not some other variables these are all the properties inside the object.

  We can destruct Arrays as well.

  [,x,y,,]=[1,2,3,4,5]
  x=2
  y=3

  We can use Rest Parameter as well
  Let [...x,,] = [1,2,3,4,5] ← fetch all in x except last 2 elements.

- **Easy variable swapping -**
  [x , y] = [y , x]

- **Classes-**

```
class ClassName {
        constructor(name) {
                this._name = name;
        }

        get name(){ ← getter
                return this._name;
        }
        set name(){ ← setter
                this._name=name;
        }
        static staticFunction(name){ ← static methods
                return new ClassName(name);
        }
        someFunction(){
                //parent_body
        }
}
```

```
class ClassChild extends ClassName {
        constructor(name, child_property) {
                super(name);
                this._child_property = child_property;
        }
        get child_property() {
                return this._child_property;
        }
        set child_property(child_property){
```

```
        this._child_property = child_property;
    }

    someFunction(){
        //childBody
    }
}
```

- **Dynamic Inheriting from classes *:***

```
function getClass(classType) {
    if(classType == 1) {
        return class1;
    } else {
        return class2;
    }

}
```

Class newClass extends getClass(2) ← calls function and gets the class and then extends it {

```
    // class body;
}
```

- **Symbols :**

Sort of like an enum and it is **immutable.**

```
let sym = Symbol("something");

// create a null object as

let x = {}
x[sym] = "new thing";
```

To see information about a symbol do -

```
sym.toString();
```

Symbols do not conflict with existing symbols

```
i.e. let x = Symbol('ashish')
     let y = Symbol('ashish')
     z ='ashish'
    Obj = {}
    Obj[x]='1'
    Obj[y]='2' ← 2 is stored somewhere else not over the 1
    Obj[z]='3'
```

Even for the same values Symbols are allocated different memory so they do not conflict.

```
i.e. assert.notEqual(Symbol(),Symbol());
```

But they can be shared as well !

assert(Symbol.for('foo'), Symbol.for('foo'));

If symbols are used as a key in an Object then it won't be listed in Object.getOwnPropertyNames
Instead it will be listed in Object.getOwnPropertySymbols.

To know which symbols are unique we can use -

Symbol.keyFor(Symbol_Object) which will return "undefined" if it is made using Symbol.for( ) instead of Symbol( )

These can be used for extending objects. As they never conflict with existing string keys.

Symbols are not private they can all be get using Object.getOwnSymbols()

- **Arrays :**
  Array.of(1,2,3); ← makes a new Array

  Array.from([1,2,3,4], (v)=> v*2) ← gives out [2,4,6,8]

  We can cycle through whole thing using :

  for (let x of Array) { }

- **Sets :**
  No duplicate values.
  let x = new Set();
  x.add(10);
  x.add('String here');
  x.has(10); ← checks if there
  x.delete(10); ← deletes 10 if exists else gives false boolean.

- **Maps :**
  let x = new Map();
  x.set('key1','val1');
  x.set('key2',10);

  x.get('key1');
  X.size

  Iterate as

  ```
  x.forEach((k,v)=> {
          console.log(k,v);
          }
  }
  ```

- **Promise :**
  Has State fulfilled, rejected or settled.
  // **Immediately fulfilled Promise.**

```
let  x = Promise.resolve(Value_to_be_resolved);
x.then((res) => console.log(res));
```

// **Later Fulfilled promise**

```
let  x = new Promise((resolve,reject) {
                setTimeout(() => resolve('Resolve Me'),2000)
        });
x.then ((res) => console.log(res));
```

// **Later promise**

```
let  x = new Promise((resolve,reject) {
                setTimeout(() => resolve('Resolve Me'),2000)
        });
x.then ((res) => console.log(res));
```

// **Rejected Promises**

```
let  x = new Promise((resolve,reject) {
                If (someCondition)
                        resolve('good value');
                 else
                        reject('bad value');
        }
x.then((val)=> console.log(val),(err)=>console.log(err));
```

// **Rejected Promises with Exceptions**

Instead of rejecting a value throw new Error("Error Message Here");
Catch it in the then chain as -

x.then(res=>console.log(res)).catch(err => console.log(err));