Object

Object comes in 2 forms : **literal** and **constructed** form

```
const litteralObj = {
  key:value
}

const constructedObj = new Object()
constructedObj.key = value;
```

They result in the same object, the only difference is how you add properties. The usual prefered way is litteral

^Types and subtypes

There are 6 types of primary types in JS. Object is one of them :

- string
- number
- boolean
- null
- undefined
- object - which has subtypes usually referred as built in Objects
  - `Function`, `Object`, `Regex`, `Boolean`, `Array`, `Number`, `Date`, `Error`, `String`

These subtypes have the appearance of being specific types / classes (similar to other language like c#) but in fact they are just built in function

```
const strPrimitive = 'I am a string'
typeof strPrimitive; // 'string'
strPrimitive instanceof String // false

const strObject = new String('I am a String')
typeof strObject; //'object'

//inspect the object sub type
Object.prototype.toString.call(strObject); //[object String] - it reveals that strObject was created by String constructor
```

The primitive value `I am a string` is not an object - it's a **primitive litteral and immutable value**. To perform operation on it such as `.length`, we need a `String object`. Fortunately JS automatically coerces a string primitive to a String object when necessary.

- `Number` and `Boolean` behave the same as `String`
- whereas `Date` can only be created by their constructed object form as it has no litteral form counterpart.
- `Object` , `Arrays` , `Function` and `Regex` are all object regardless of wether the litteral or constructed form is used

Content

The content of an object consists in values stored at specific named locations, which we call properties.

Technically, the object **does not contain** the values themselves, they just store the property names which are pointers (`references`) to where the values are stored.

2 ways to access the content

```
const myObj = {
  a: 2
}

myObject.a //2
myObject[a] //2
```

Property Descriptors

With ES5, all properties are described in terms of a `property descriptor`

```
const myObj = {
  a:2
}

Object.getOwnPropertyDescriptor(myObj,"a")
// {
//  value:2,
//  writable: true;
//  enumerable: true,
//  configurable: true
// }


Object.defineProperty(
  myObj,
  "b",
  {
    value:3,
    writable:true,
    configurable:true,
    enumerable:true
  }
)
myObj.b // 3
```

- writable : descriptor that allows/doesn't allow you to override the value
- configurable : allows / doesn't allow to override the descriptors themselves
  - changing configurable to `false` is therefore a one way operation.
  - `configurable:false` prevents you also from deleting a property
- enumerable : determines if the property will be shown in object property enumeration like `for..in`

Immutability

Sometimes you want to make object that cannot be changed. ES5 adds support for doing that in a variety of ways. All of them create `shallow` immutability, meaning that any reference to other objects like `arrays`, `objects` or `function` will remain mutable.

Object constant

To create a property that cannot be changed,redfined or deleted - you can combine `writable: false` and `configurable: false`

Prevent extensions

If you want to prevent an object from receiving any new properties, but leave other properties alone - you can :

```
const obj = {
  a:2
}
Object.preventExtensions(obj)

obj.b = 3 //fails silently without strictmode, throws a typeError with it
obj.b // undefined
```

Seal

`Object.seal(...)` takes an existing object and calls `Object.preventExtensions` while also marking all properties as `configurable: false` resulting in an object where you can no more add new properties, deleting existing ones, neither change its descriptors. You can still modify existing values.

Freeze

`Object.freeze(...)` basically takes an existing object, `Object.seal()` it and mark all properties as `writable:false` preventing the change of any existing values

Since immutability has become more important over the last years with React and Redux - There are several libraries that wraps around those methods.

Getters and Setters

To access properties or set properties values - JS relies on `[[Get]]` or `[[Put]]`

ES5 introduced a way to override those default operations at a per-property level through the use of `getters` or `setters`

When you define a property to have either a getter or a setter, its definition becomes an `accessor descriptor` as opposed to a `data descriptor`

```
const myObject = {
  //define a getter
  get a() {
    return 2
  }
}

myObject.a = 3;
myObject.a //2
```

We created a property on the object that actually doesn't hold a value but whose access automatically results in a hidden function call to the getter function, with whatever value it returns being the result of the property access.

To be able to set the value you need a `setter` :

```
const myObject = {
  //define a getter
  get a() {
    return this._a_
  }
  set a(val) {
    this._a_ = val * 2
  }
}

myObject.a = 2;
myObject.a // 4

myObject //{_a_ : 4}
```

Existence and enumeration

```
const myObject = {a:2}

("a" in myObject) //true
("b" in myObject) //false

myObject.hasOwnProperty("a") //true
myObject.hasOwnProperty("b") //false
```

Difference :

- the `in` operator goes up in the *Prototype* chain werheas `hasOwnProperty` checks if the property exists directly on the object

When you set `enumerable` property descriptor to `false`, the property still exists on the object itself but **wont be included if the object are iterated through** like in a `for...in` loop

Iteration

`for in` loop iterates over the list of enumerable properties on an object - **including its *prototype* chain** meaning that you only get the properties identifiers and then access the value through the property identifier.

ES6 `for...of` in contrast iterates over the value itself. This helper asks for an iterator object. Arrays have a built one, while object itentionnaly miss one. But you can add one yourself.

```
const myArray = [1,2,3]
const it = myArray[Symbol.iterator]();

it.next()//{value:1, done:false}
it.next()//{value:2, done:false}
it.next()//{value:3, done:false}
it.next()//{done:true}
```