Objects

Objects are real life entities which are used to represent data. Objects are collection of key value pairs. Multiple highly related properties can be encapsulated inside an object. We can also say that the object is combination of related key value pairs e.g.

```
let person = { name : 'anubhav', age : 21 };
```

Accessing properties of objects

There are two main ways of accessing properties of objects
Dot Notation or Bracket Notation

```
person.name or person['name']
```

Updating properties of objects: Objects are mutable

```
person['name'] = 'shanu';
```

Testing Objects for Properties

```
person.hasOwnProperty('name')
```

Array of Objects

```
var cats =
[
    {
        name : "Smile",
        color : "black"
    },

    {
        name : "kattie",
        color : "white"
    }
];

console.log(cats[0]); //{ name: 'Smile', color: 'black' }
```

```
const circle = {
    radius: 1,
    draw : function()
    {
        console.log('draw');
    }
}
```

Imagine we want to create two circle objects. We can copy the above code but the problems are:

1. Duplicate draw method
2. If we bug in a function then we have to fix it in multiple copies.
3. If we have multiple other methods, we will end up with redundant code.

Factory Functions

Just like factory producing products, these factory functions produce objects.

```
function createCircle(radius)
{
    return {

        radius: radius,
        draw : function()
        {
            console.log('draw');
        }
    }
}
```

or

```
function createCircle(radius)
{
    return {

        radius: radius,
        draw()
        {
            console.log('draw');
        }
    }
}

const circle1 = createCircle(1);
const circle2 = createCircle(2);
```

Now we have different circles with different radii but same draw function defined at one place only. For naming of factory functions, we use camel case.

Constructor Functions

```
function Circle(radius)
{
    this.radius = radius;
    this.draw = function()
    {
        console.log('draw');
    }
}

const circle1 = new Circle(1);
const circle2 = new Circle(1);
```

When we use the new operator three things happen under the hood.

1. The new operator creates an empty JavaScript object.
   `const circle1 = {};`

2. New operator will set this to point to circle1.
3. Finally new operator will return this object from constructor function.
   `return this;`
   We don't need to add this statement explicitly.

Dynamic Nature of Objects

We can add or remove properties after creating them.

```
const circle = { radius: 1 };

circle.color = 'red';
circle.outline = 'black';
delete circle.color;
```

Constructor Property

Every Object in JavaScript has a constructor property that refers to the function that was used to create that object.

```
const circle = { radius: 1, backgroundColor: 'red', outline: 'black' };
console.log(circle.constructor); // ƒ Object() { [native code] }

const name = 'Anubhav';
console.log(name.constructor); // ƒ String() { [native code] }
```

```
const isApproved = true;
console.log(isApproved.constructor); // ƒ Boolean() { [native code] }

let x = 5;
console.log(x.constructor); // ƒ Number() { [native code] }
```

When we create an object using the object literal syntax internally the JavaScript Engine uses this constructor function.

Functions as Objects

```
function Circle(radius)
{
    this.radius = radius;
    this.draw = function()
    {
        console.log('draw');
    }
}
```

We can access all the properties using the dot notation.

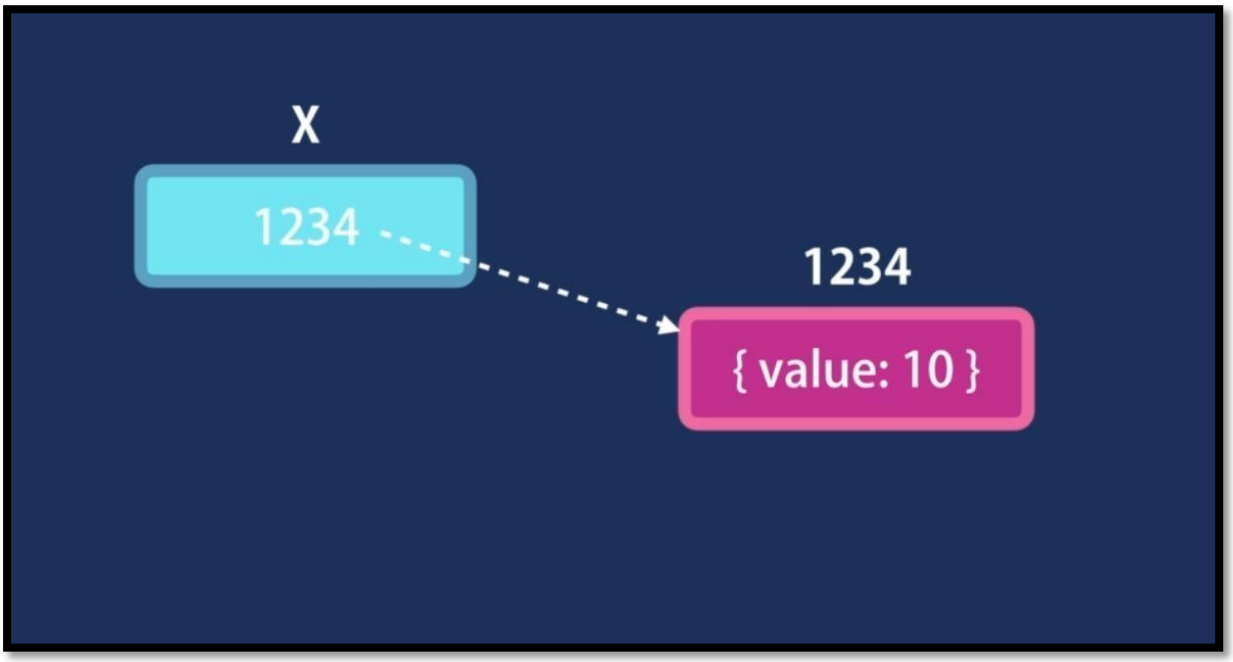
```
console.log(Circle.name); // Circle
console.log(Circle.length); // 1

const Circle1 = new Function('radius', `
    this.radius = radius;
    this.draw = function()
    {
        console.log('draw');
    }`
);

const circle = new Circle1(2);
```

Primitives vs References

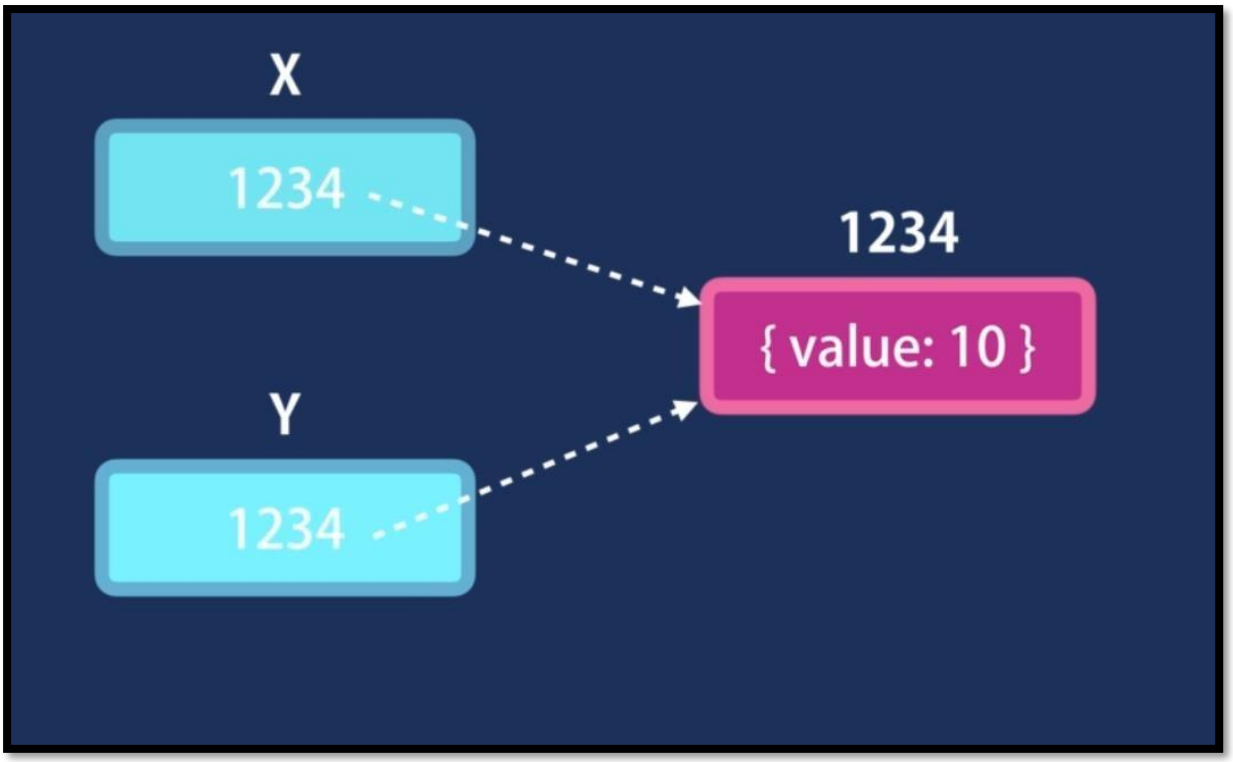
```
let x = 10;
let y = x;

x = 20;

console.log(y);
```

Here x and y are independent of each other.

```
let x = {value: 10};
let y = x;

x.value = 20;

console.log(y); // { value: 20 }
```



Here, object x is not stored in object y instead the memory address of object x is stored in the object y. That's why both x and y point to same object.



When we change the object either using the x or y the changes are immediately visible to another variable.

Primitives are copied by their **value**

Objects are copied by their **reference**

Cloning an Object

```
const person1 = { name : 'Anubhav', age: 15 };

const person2 = {};

for(let key in person1)
{
    person2[key] = person1[key];
}
```

Using Object.assign();

```
const person2 = Object.assign({}, person1);
```

Using Spread Operator

```
const person2 = {... person1};
```

Adding New Properties While Cloning An Object

```
const obj = Object.assign( { prop1 : value1 }, targetObj ) ;
```

Garbage Collector

The JavaScript Engine has garbage collector. It finds the variables and constants that are no longer used and then deallocates the memory that was allocated earlier.

String Template Literals

```
const mail = `Hi, John${2},

Thank you for joining my mailing list.

Regards,
Anubhav Gupta.`;

console.log(mail);
```

Enumerating Properties

```
const person = { name : 'Anubhav', age: 15 };

for(let key of Object.keys(person))
{
    console.log(key, person[key]);
}

//name Anubhav
//age 15
```

```
const person = { name : 'Anubhav', age: 15 };

for(let entry of Object.entries(person))
{
    console.log(entry[0], entry[1]);
}

//name Anubhav
//age 15
const person = { name : 'Anubhav', age: 15 };

if('name' in person)
{
    console.log('Yes'); //Yes
}
```

Math Object

It contains properties and methods for mathematical constants and functions.

```
Math.E //returns Euler's Number
Math.PI //returns PI
Math.round(4.9) //returns 5
Math.abs(-256) //returns 256
Math.pow(2, 5) //returns 32
Math.floor(3.999) //returns 3
Math.ceil(3.999) //returns 4
Math.min(1,2,9) //returns 1
Math.max(1,2,9) //returns 9
Math.random() //returns random number between 0 and 1.
Math.floor(Math.random() * (max - min) + min);
//returns random integer between max and min.
```

freeze

The Object.freeze() static method freezes an object. Freezing an object prevents extensions and makes existing properties non-writable and non-configurable. A frozen object can no longer be changed: new properties cannot be added, existing properties cannot be removed, their enumerability, configurability, writability, or value cannot be changed, and the object's prototype cannot be re-assigned. freeze() returns the same object that was passed in.

```
Object.freeze(obj)
```

Timing Events

```
setTimeout(()=>
{
    console.log('Hello');
}, 2000)


var i = 0;
var id = setInterval(()=>
{
    console.log('Hello');
    i++;
    if(i == 6)
    {
        clearInterval(id);
    }
}, 2000)

//Hello
//Hello
//Hello
//Hello
//Hello
//Hello
```

object vs Object

object is the type of value while Object is the constructor function.

Shallow Copy vs Deep Copy

```
//shallow copy
const person1 = { firstName: 'John', lastName: 'Doe', hobbies: ["Singing",
"Dancing", "Blogging"] };
const person2 = {...person1};
person2.hobbies[1] = "Cooking";

console.log(person1); // { firstName: 'John', lastName: 'Doe', hobbies: [
'Singing', 'Cooking', 'Blogging' ] }
console.log(person2); // { firstName: 'John', lastName: 'Doe', hobbies: [
'Singing', 'Cooking', 'Blogging' ] }

//Deep copy
const person1 = { firstName: 'John', lastName: 'Doe', hobbies: ["Singing",
"Dancing", "Blogging"] };
const person2 = JSON.parse(JSON.stringify(person1));
person2.hobbies[1] = "Cooking";

console.log(person1); // { firstName: 'John', lastName: 'Doe', hobbies: [
'Singing', 'Dancing', 'Blogging' ] }
console.log(person2); // { firstName: 'John', lastName: 'Doe', hobbies: [
'Singing', 'Cooking', 'Blogging' ] }
```