Object Oriented Programming In JS

In software engineering, anything that we have can be mapped to a blueprint or a vision. For example, in Amazon, every product has a similar structure, we can generalise this and make sure we don't repeat anything again and again.

Classes

Classes are blueprint on a set of real life entities.
Classes are going to represent how an entity should look and behave. When I say how do they look, I refer to the properties they posses. And when I say how they behave then I mean what actions can be performed on them.

Let's take an example:

Consider a `Product` class, inside this product class what do you think should be the properties of a product ? What is a property ? Property is just any feature that the product can contain.

- name -> Name of the product
- price -> Price of the product
- company -> Company the product belongs to
- reviews -> set of feedbacks / reviews the product got
- rating -> Rating of the product
- description -> Description of the product.\
- and more ...
  So above are the set of properties that a Product class can posses. In technical terms these properties are referred as `Data Members`. If two products are different from each other then atleast one of the data members should have a different value in them.

Now, what are the things we can do with a product ?

- We can buy a product
- Add product to cart
- We can display the details of a product
- We can wishlist a product
- We can rate a product
- We can add review to the product

- and more ....
  These are the actions which can be done on a product, which we can also consider as the behaviour of the product. And in technical terms these are referred as `Member functions`.

How to make a class in JS ?

In JS, we have a `class` keyword, that can help us to make a class and include the data members and member functions inside it.

Every class has a name associated to it as well, that we have to define once we declare the class.

```
class NameOfTheClass {
        // you can details like member functions and data member
}
```

Let's write a product class:

```
class Product {
        name;

        price;

        category;

        description;

        rating;



        addToCart() {

                console.log("Product added to cart");

        }



        removeFromCart() {

                console.log("Product removed from cart");
        }



        displayProduct() {

                console.log("Product displayed");

        }



        buyProduct() {

                console.log("Product bought");

        }



}
```

constructor : a very special member function of every class

Every class that we make in JS, has one special member function called as constructor. What is so special about it ?
So whenever we create an object using classes, the constructor is first method that is automatically called by JS. It is already available for any class you make. The default version of this constructor is also called `default constructor`.
We can change the implementation of the default constructor by writing one of our own. How ? By doing something like this:

```
class Product {

        constructor() {
                // This is your custom constructor
        }
}
```

We will discuss more internals of constructor later.

Objects

Using classes the final entity that we will develop is called as `objects`. How can we create an object of a class ?
There is a keyword in JS called as `new` which can help us to create an object out of a class. Don't map the use case of `new` keyword in JS with other languages, it is very different from other languages.

```
let iphone = new Product();
```

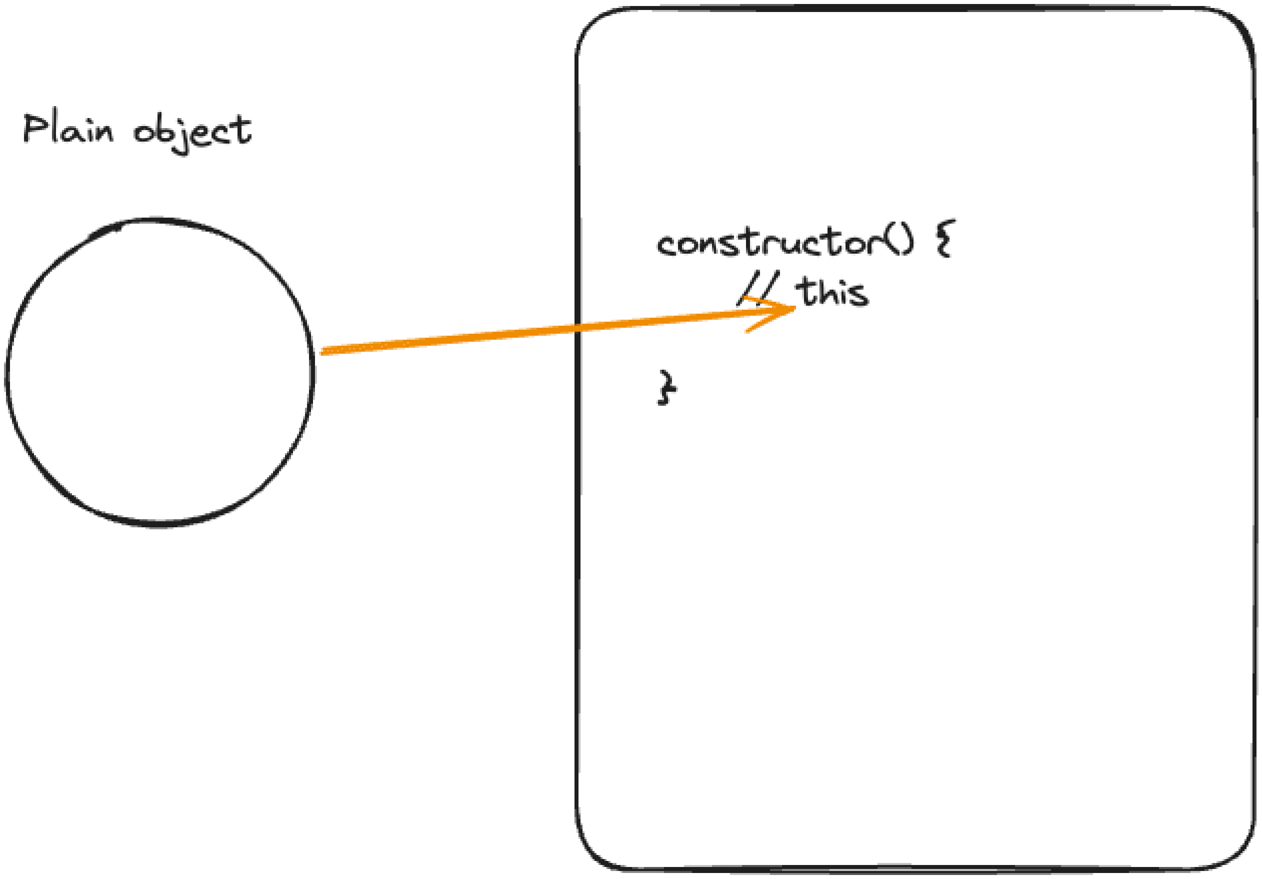
So, to create an object we write `new` and then mention the name of the class followed by a pair of parenthesis.

How new works ?

Every time we call new, it does the following 4 step procedure:\

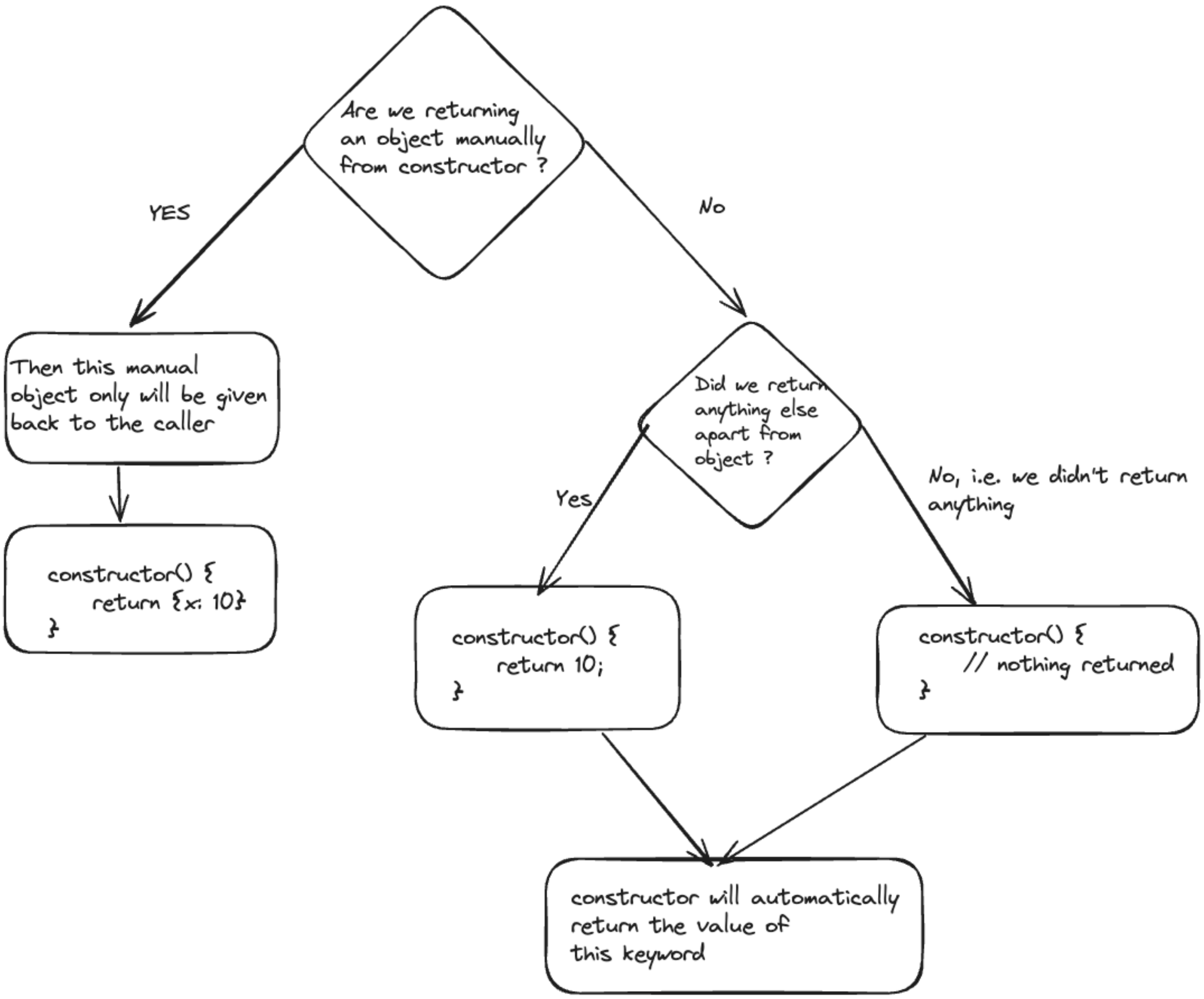
1. It creates a brand new plain & absolutely empty object.
2. It calls the constructor of the class and passes the newly created object (not as a parameter) but inside a keyword called as `this`. So constructor automatically has access to the this keyword and when we call new , then the this keyword has access to the plain object created in step 1 and constructor now can use the this keyword inside it. And then

whatever is logic of constructor it is executed.



3. In step 3, new triggers everything need to be done for prototypes to work (will discuss later).
4. Now, if from a constructor an object is manually returned then this manual returned object is stored in the called variable, otherwise in any other case i.e. either we don't return anything or return something apart from object, constructor doesn't care about it and

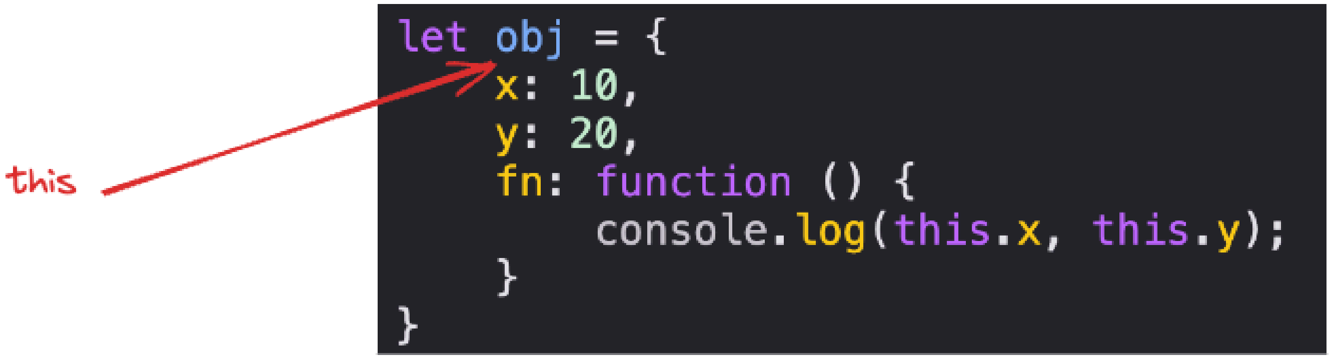
returns the value inside the this keyword.


Are we returning an object manually from constructor ?
YES
No
Then this manual object only will be given back to the caller
constructor() {
return {x: 10}
}
Did we return anything else apart from object ?
Yes
No, i.e. we didn't return anything
constructor() {
return 10;
}
constructor() {
// nothing returned
}
constructor will automatically return the value of this keyword

this keyword of JS

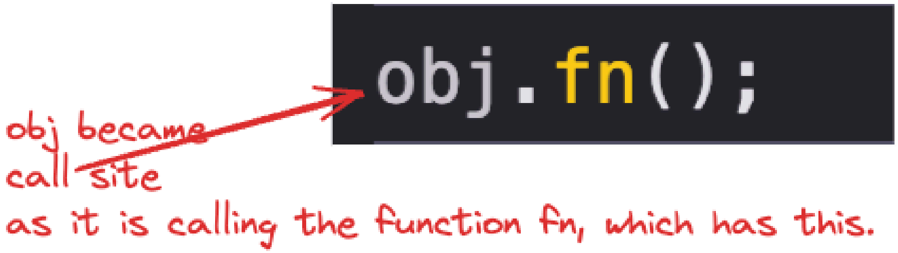
- this in JS works very differently than other languages.
- this as a keyword is available to be accessed in any function or even outside any function, and in classes as well.
- If we can use `this` keyword anywhere, then what's the value stored inside `this` ?

In most of the cases this refers to the call site.
Ahhh! What is a call site ?
Call site can be an object, or a position or may be even the new keyword. It refers to the entity which is calling this keyword.



this

```
let obj = {
    x: 10,
    y: 20,
    fn: function () {
        console.log(this.x, this.y);
    }
}
```



```
obj.fn();
```

obj became
call site
as it is calling the function fn, which has this.

It has an exception, this keyword will not refer to the call site if used inside an arrow function
In case of arrow functions, this is resolved using lexical scoping.

```
let obj = {
    x: 10,
    y: 20,
    fn: function() {
        const arrow = () => {
            console.log(this.x, this.y);
        }
        arrow();
    }
}
```

In this code, this is present inside the arrow function, hence we will resolve it lexically.

1. Is `this` defined in the scope of arrow function ? No
2. We go one scope up, i.e inside function fn.
3. Is `this` defined inside fn? yes, because fn is a normal function, we have a definition of this inside it which is the call site
4. Who is the call site ? Obj object which is responsible to trigger fn is the call site
5. Hence `this` refers to obj object and when we call arrow function we get output as `10 20`.

That means, when we make a new object using the `new` keyword, then `new` keyword creates a plain object and this plain object is the call site for the constructor hence, this keyword refers to the plain object altogether.

Problems with current class implementation

In the current implementation, we don't add any security on the updation of the values of the data members in the class.
After object creation, anyone can come up and update the object of our class without any issues.
This violates a fundamental principle of OOPs, called as `Encapsulation`. It says that, classes are a bundle of implementation in themselves. In any case we should not be required to modify the data members of a class outside the class. Why, because then our data members will be allocated any random value without any issue.
To resolve this, we can hide out data members from being accessed outside of the class. How ?
In JS, we can make our data members private i.e. the data members will be accessible for read and write only inside the class.
To make a data member private, we just prepend a `#` before it's declaration.

```
class Product {
        #name;
        #price;
        #description

        ...
}
```

Here, name price and description are now private, they cannot be accessed outside of the class. But we can use it inside the class properly.

```
class Product {
        #name;
        #price;
        #description

        displayProduct() {
                console.log(this.#name, this.#price, this.description); //
we are accessing inside the class
        }
}
```

Now to make sure that users are able to get and set the values of our data members, we write getter and setter methods. These getter setter methods will help us to write validation logic so that no one can allocate random values to our data members.

```
class Product {
        #price;

        getPrice() {

                return this.#price;

        }



        setPrice(p) {

                if (p > 0) {

                        this.#price = p;

                } else {

                        console.log("Invalid price");

                }

        }
}
```

There is an alternative way to defined getter and setter in JS, we can use the get and set keywords to handle this.

```
class Product {

        #description;

        get description() {
                return this.#description;
        }
        set description(d) {
                if(d.length === 0) {
                                console.log("invalid desc");
                                return;
                }
                this.#description = d;
        }
}
```

So here we have defined a getter by putting the get keyword with the property name whose getter we want to define and set with the property name whose setter we need to define. Both are function and in fact setter function also takes an argument which it will attempt to set in the property.

How to call them ?

```
const iphone = new Product();
iphone.description = "Something";
console.log(iphone.description);
```

`iphone.description = "Something"` will help us to access the setter and pass `Something` as an argument to the setter function.
`iphone.description` will help us to call the getter method defined.

Class as wrappers on function

In JS, every class we write is a wrapper over functions, in older versions of JS `class` keyword didn't exist hence to do the blue printing we used to have only functions.

```
function Product(n, p, d) {
    this.name = n;
    this.price = p;
    this.description = d;

    this.displayProduct = function() {
        console.log("Name:", this.name, "Price", this.price, "description:",
this.description);
    }
}
```

Now the above Product function acts as a `Function constructor`, here it takes a few parameters and uses `this` keyword to assign them. To call this constructor we do:

```
let iphone = new Product("Iphone 11", 900, "Apple iphone 11");
```

Now, here we are calling function constructor of Product with new keyword. And new keyword will do the following 4 things:

1. Create an empty object.
2. Assign the `this` keyword in the function constructor to the new empty object and call the constructor.
3. Do prototyping.
4. If the constructor returns a new object manually, return the same from the constructor, else if no object is manually returned or a non obejct value is returned then we return this from the constructor.
   In step 2, when we execute the constructor, we assign data members like name, price etc and member functions like `displayProduct`.

Builder Design Pattern

Assume we have a class like this:

```
class Product {
        #name;
        #price;
        #category;
        ....
        constructor(name, description, price, category, discount  ..... ) {
                this.name = name;
                this.price = price;
                this.description = description;
                ....
        }
}
```

We have a big problem here that, the constructor has too many params and when calling the constructor we have to remember which params are going to have which value in the right sequence.
Let's say we want to have different ways of object creation, for example we want to create objects with only name and price, or may be with name, price and category, or may be with just name and so on, then we cannot have multiple constructors in JS.

To avoid this we can use the default constructor and instead of using new constructor to allocate values we can use getter setters.

```
class Product {
        #name;
        #price;
        #category;
        ....
        get name() { .. }
        set name(name) {  ...  }
        get price() {...}
        set price(p) {...}
}
```

If we use this approach then we cannot apply validation to the object before it's creation. Due to any reason if we want a scenario where due to some validations not being fulfilled we have to stop the creation of object then it is not possible here, because to use getter setters we have to first create the object and then only we can get or set the values.

How to solve both the issues ?

We need getter setters for good encapsulations, because we might need to for updating values of getting values. And to avoid the validation issue, we need custom constructor also. Constructors allow us to validate object before it's creation.

A nice way would be to pass a single object in the constructor parameter, this object will be having all the keys as properties of the class and values as the value we want to assign them, if we don't want to assign a few values we can keep those keys as undefined. Let's call this object as `builder`.

```
class Product {
        #name;
        #price;
        #category;
        ....
        constructor(builder) {
                this.name = builder.name;
                if(builder.price > 0)
                        this.price = builder.price;
                this.description = builder.description;
                ....
        }
        get name() { .. }
        set name(name) {  ...  }
        get price() {...}
        set price(p) {...}
}
const p = new Product({name: "iphone 11", price: 8700, description: "Apple
iphone"});
```

This is the v1 of our Builder patter, this pattern helps us to create objects in a cleaner way.

We can enhance this implementation as well. To enhance it we are going to introduce a new getter called as `Builder`. You can call this as getBuilder, newBuilder etc.

~~~
class Product {
        #name;
        #price;
        #category;
        ....
        constructor(builder) {
                this.name = builder.name;
                if(builder.price > 0)
                        this.price = builder.price;
                this.description = builder.description;
                ....
        }
        get Builder() { ... }
        get name() { .. }
        set name(name) { ... }
        get price() {...}
        set price(p) {...}
}```

This implementation has a problem now. This Builder getter function will be
used to create the builder object that we passed in the constructor. But to
access this Builder getter function we need an object. It doesn't make sense
to create an object to fetch this builder getter and then use it to create
the final object.

That's where `static` keyword comes into the picture.

### static keyword
This keyword will help us to associate any method or property to a **class**
instead of an object. We don't need to initialise an object to access a
static property / method. When the class is loaded in the memory, static
properties and methods are created then and there.


SO, now we can make our Builder getter static so that we can access it
directly using Product class.

```JS
class Product {
        #name;
        #price;
~~~

~~~
        #category;
        ....
        constructor(builder) {
                this.name = builder.name;
                if(builder.price > 0)
                        this.price = builder.price;
                this.description = builder.description;
                ....
        }
        static get Builder() {
                class Builder {
                        ...
                        // this class will implement the Builder object related things
                }
        return new Builder(); // when someone calls the Builder getter, they will get a new Builder object.

        }
        get name() { .. }
        set name(name) {  ...  }
        get price() {...}
        set price(p) {...}
}```

when someone calls the Builder getter, they will get a new Builder object.

```JS
Product.Builder; // this gives us new Builder object
~~~

Now in the builder class we have a constructor, which is not mandatory but we have put it so that we can allocate default values to all properties of product class, but it's not mandatory. Apart from this it has some setter methods.

```
class Product {
        #name;
        #price;
        #category;
        ....
        constructor(builder) {
                this.name = builder.name;
                if(builder.price > 0)
                        this.price = builder.price;
                this.description = builder.description;
                ....
```

~~~
        }
        static get Builder() {
                class Builder {
                        constructor() {
                                this.name = ""; // this name property belongs to the builder
                                this.price = 0; // this price property belongs to the Builder
                                // setting default values
                        }
                        setName(incomingName) {
                                this.name = incomingName;
                        }
                        setPrice(p) {
                                this.price = p;
                        }
                }
        return new Builder(); // when someone calls the Builder getter, they will get a new Builder object.

        }
        get name() { .. }
        set name(name) {  ...  }
        get price() {...}
        set price(p) {...}
}```

But having setter like this has a problem, when we will call it, we have to again and again use the builder object to call them.
```JS
const b = Product.Builder;
b.setName("iphone11");
b.setPrice(1100);
 ...
~~~

To improve this we can return the builder object from every setter. Using this mechanism we can avoid putting builder object again and again in function calls, and instead chain the functions calls.

```
class Product {
        #name;
        #price;
        #category;
         ....
        constructor(builder) {
```

~~~
                this.name = builder.name;
                if(builder.price > 0)
                        this.price = builder.price;
                this.description = builder.description;
                ....
        }
        static get Builder() {
                class Builder {
                        constructor() {
                                this.name = ""; // this name property belongs to the builder
                                this.price = 0; // this price property belongs to the Builder
                                // setting default values
                        }
                        setName(incomingName) {
                                this.name = incomingName;
                                return this; // to return builder object we can use this keyword
                        }
                        setPrice(p) {
                                this.price = p;
                        }
                }
        return new Builder(); // when someone calls the Builder getter, they will get a new Builder object.

        }
        get name() { .. }
        set name(name) {  ...  }
        get price() {...}
        set price(p) {...}
}```

Now to call the Builder, we can say
```JS
Product.Builder
.setName("Iphone")
.setPrice(1100)
.setCategory( ... )
~~~

But still we only have the builder object, not the Product object. To create a product, we can create a build method inside the builder class which will call the Product constructor and then pass the builder object inside it.

```
class Product {
        #name;
        #price;
        #category;
        #discount;
        ....
        constructor(builder) {
                this.name = builder.name;
                if(builder.price > 0)
                        this.price = builder.price;
                this.description = builder.description;
                ....
        }
        static get Builder() {
                class Builder {
                        constructor() {
                                this.name = ""; // this name property belongs to the builder
                                this.price = 0; // this price property belongs to the Builder
                                // setting default values
                        }
                        setName(incomingName) {
                                this.name = incomingName;
                        }
                        setPrice(p) {
                                this.price = p;
                        }
                        build() {
                                return new Product(this);
                        }
                }
        return new Builder(); // when someone calls the Builder getter, they will get a new Builder object.

        }
        get name() { .. }
        set name(name) { ...  }
        get price() {...}
        set price(p) {...}
}```

Product constructor takes a builder object, so we passed the this. keyword
as here this points to builder object.

Now we can getn the Product object by saying:
```

~~~
```JS
const p = Product.Builder
                                .setName(..)
                                .setPrice(..)
                                .setCategory(..)
                                .
                                .
                                .build();
~~~