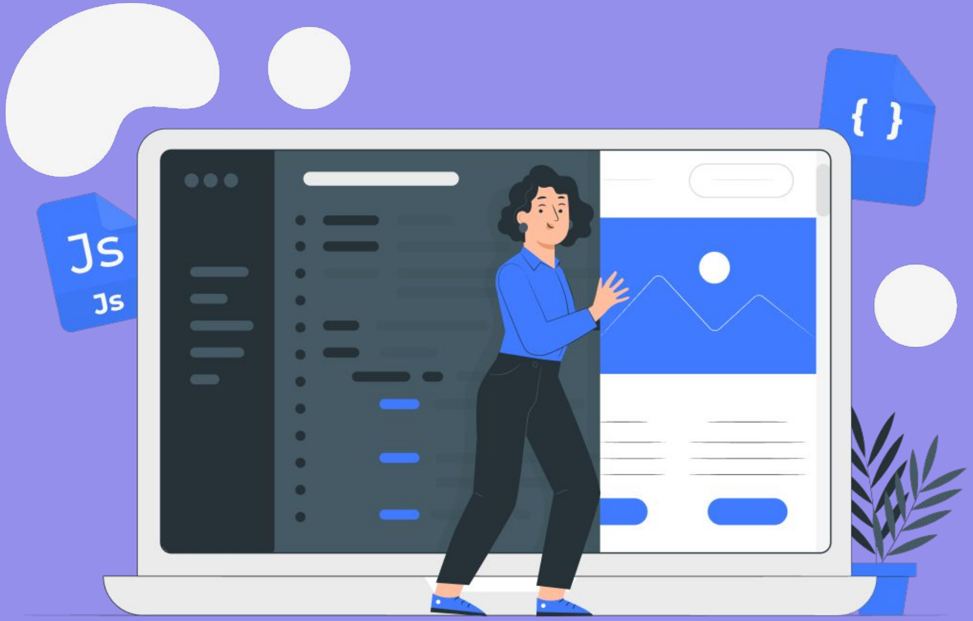
# Intro to OOPs in JS



Relevel
by Unacademy

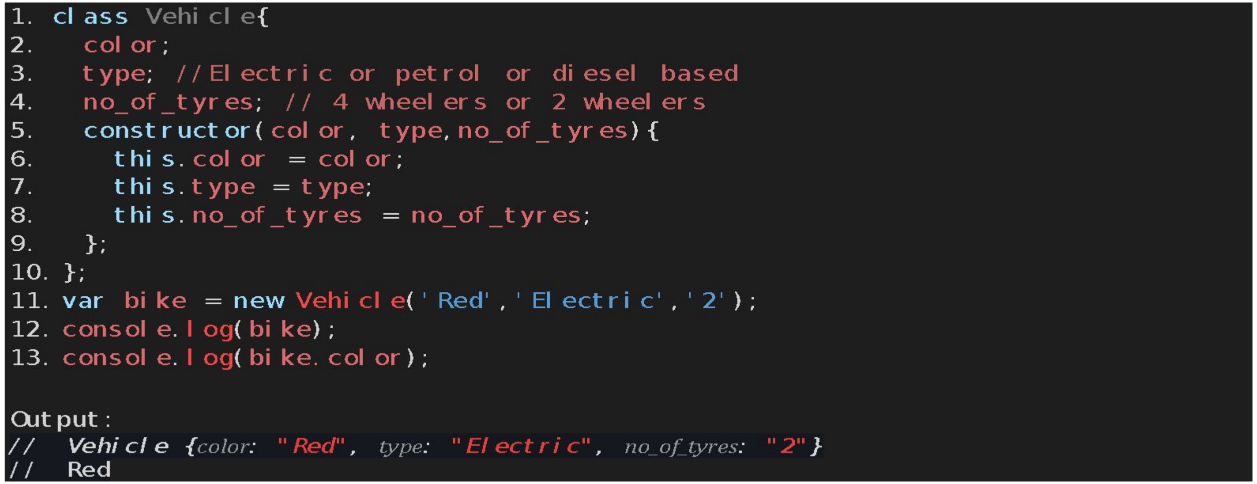
## Introduction to OOPs:

- OOPs stands for Object-Oriented Programming.
- As the name suggests, we use objects in our program to represent real-world entities. OOPs, provide a great deal of feasibility in terms of flexibility and modularity. OOPs also helps us to provide more control over our code by allowing only a certain code member to access the other as per our requirements.
- The above can be achieved by using various concepts of OOPs like Encapsulation, Abstraction, Inheritance, and Polymorphism. We will discuss these in detail in a later section.

## Object

- Object - the instance of a class.
- It is a collection of related data and functionality.
- Consist of various variables and functions.
- These variables and functions are called as properties and methods respectively in OOPs.

## Example:



```
1.  class Vehicle{
2.     color;
3.     type; //Electric or petrol or diesel based
4.     no_of_tyres; // 4 wheelers or 2 wheelers
5.     constructor(color, type,no_of_tyres){
6.        this.color = color;
7.        this.type = type;
8.        this.no_of_tyres = no_of_tyres;
9.     };
10. };
11. var bike = new Vehicle('Red','Electric','2');
12. console.log(bike);
13. console.log(bike.color);

Output:
//  Vehicle {color: "Red", type: "Electric", no_of_tyres: "2"}
//  Red
```

- Here we can see we have defined a class Vehicle and we have created an object of Vehicle class at line 11, using “new Vehicle('Red','Electric','2');”.
- In line 5, the constructor is defined by taking a few parameters, and these parameters are now passed to each property of the Vehicle class using "this" keyword.

## Constructor:

- A constructor is a function in javascript that is called when an object of a class is initialised using the “**new**” keyword.
- Constructor is used to set the values of class properties during initialisation.
- Construction can be defined only once in a class.
- Super keyword is used inside the constructor to call the constructor of the parent class. More detail on this we will see in a later section.

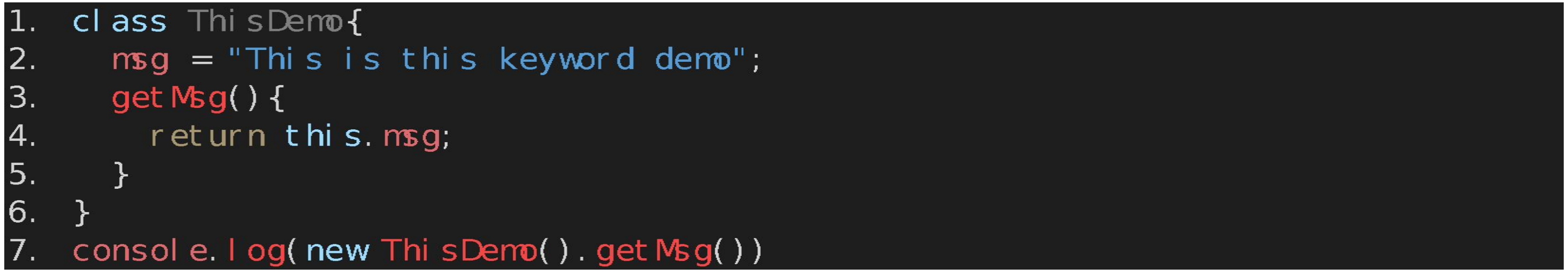
## this:

This keyword is used in javascript to refer to an object.

Which object is being referred to depends on how it's used.

When used in method, this keyword refers to the **object**

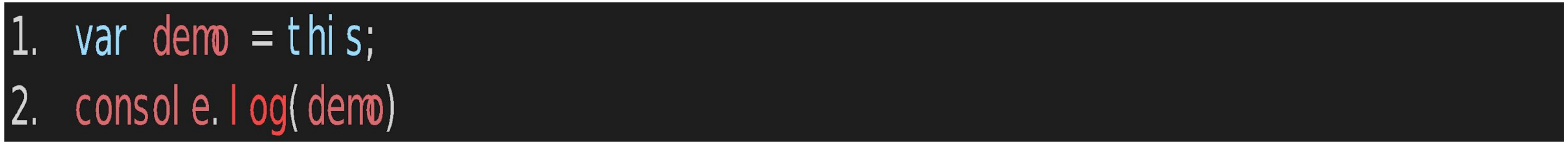
## this Example:



```javascript
class ThisDemo{
  msg = "This is this keyword demo";
  getMsg(){
    return this.msg;
  }
}
console.log(new ThisDemo().getMsg())
```

## this Example:

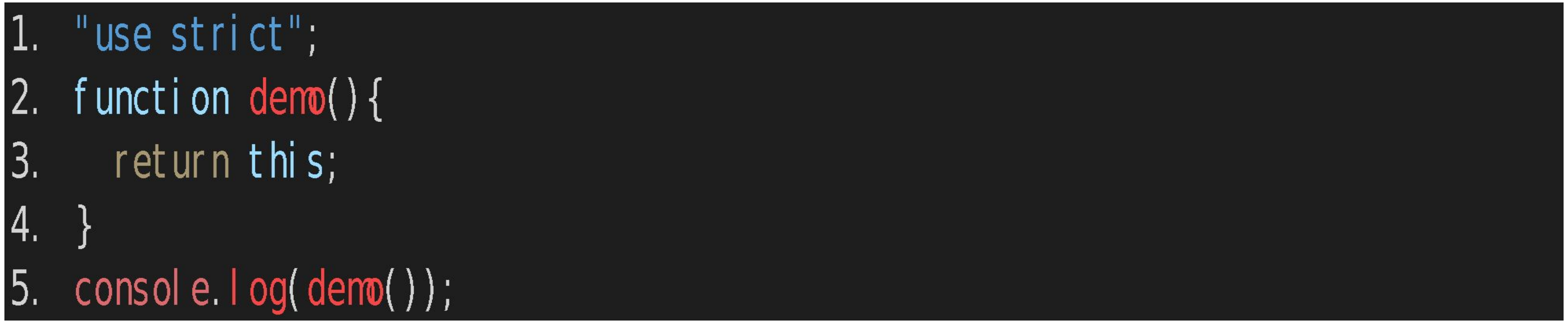
When used alone this refers to **global Object**.



```
var demo = this;
console.log(demo)
```

## this Example:

When used in “strict mode” inside function this refer to **undefined**.



```
"use strict";
function demo(){
   return this;
}
console.log(demo());
```

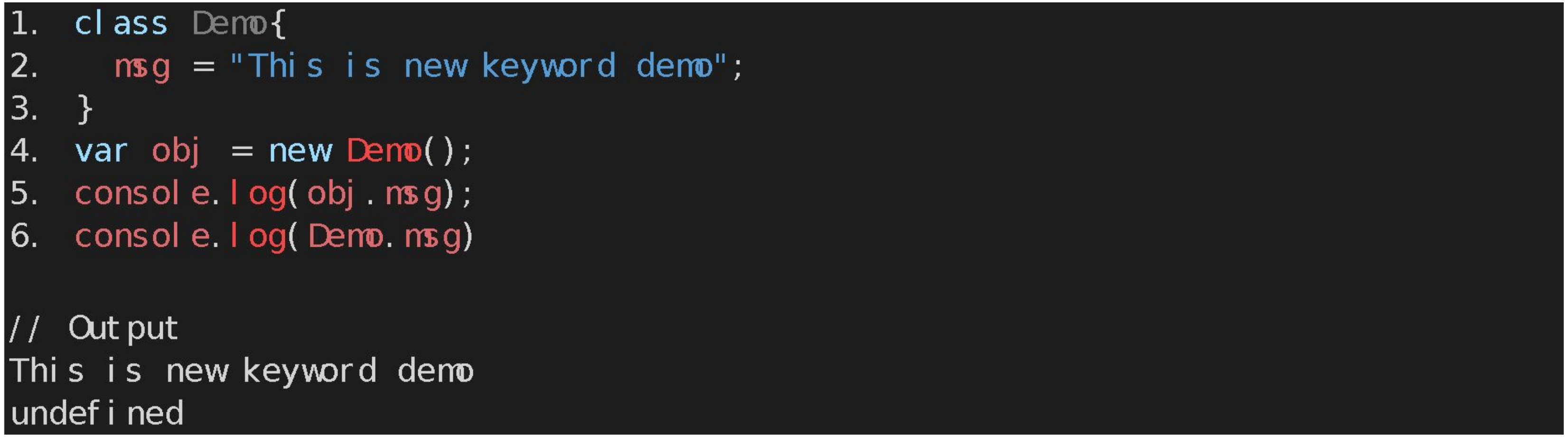
## new:

new is a keyword that is used to create an instance of a user-defined object.

Syntax:

```
new constructorName([argument list])
```

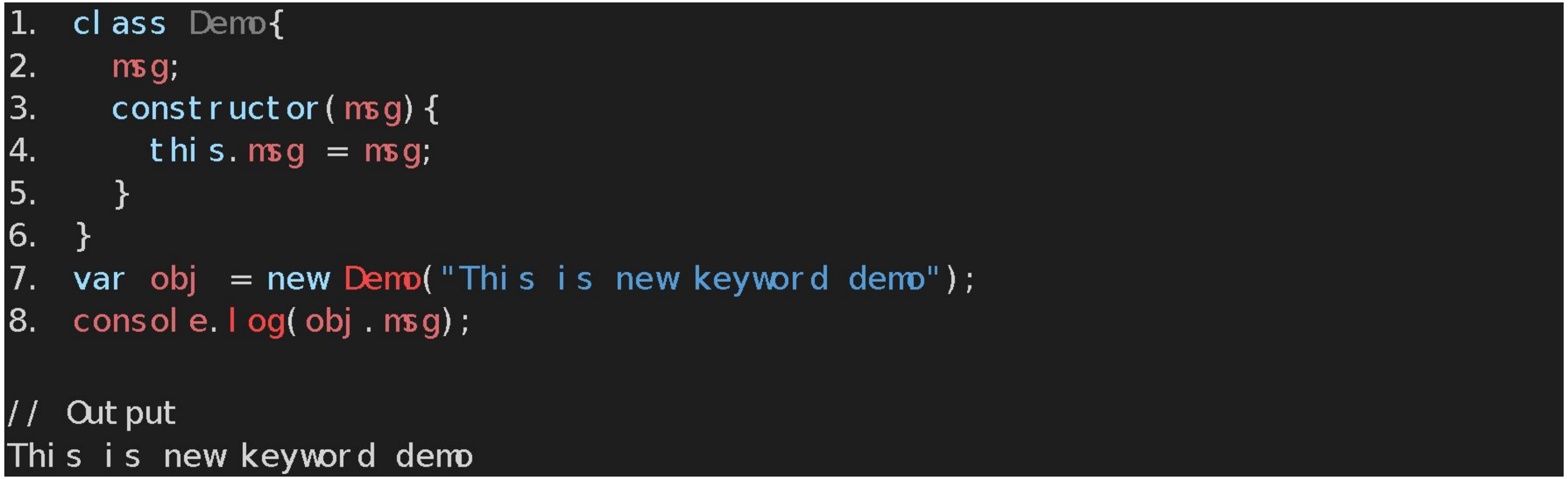
## Example:



```
1.  class Demo{
2.     msg = "This is new keyword demo";
3.  }
4.  var obj  = new Demo();
5.  console.log(obj.msg);
6.  console.log(Demo.msg)

// Output
This is new keyword demo
undefined
```

Here we are using a new keyword without any parameter.
At line 5, we try to fetch the “msg” value using object “obj”, so we got the result.
At line 6, we tried to fetch the “mdg” directly with Classname, so we got undefined.

## Example:



```
class Demo{
  msg;
  constructor(msg){
    this.msg = msg;
  }
}
var obj = new Demo("This is new keyword demo");
console.log(obj.msg);

// Output
This is new keyword demo
```

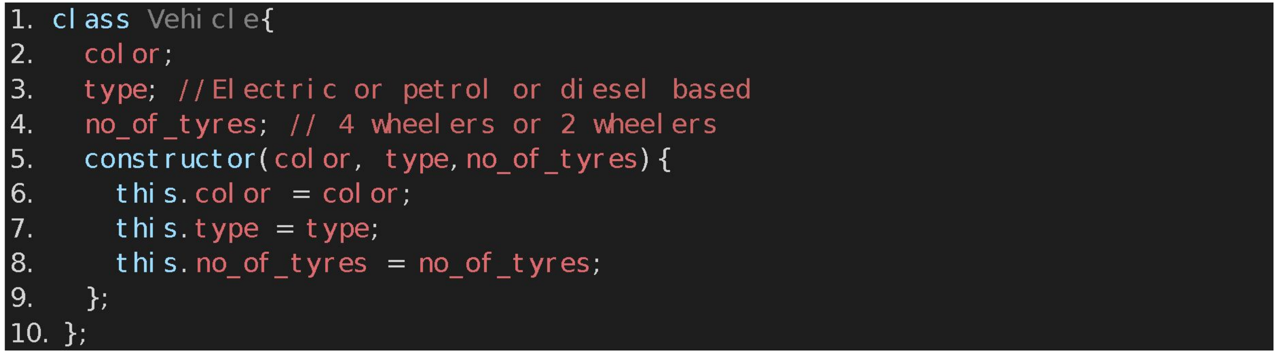
Here we have passed a parameter with new keyword.

## Class:

A class is a blueprint of a real-life entity.
It defines the characteristics of an Object like color, shape, size, etc.

## Example:

For instance, every vehicle will have a certain color, type, and tires. So if we consider the vehicle as class, then we can show it programmatically as below,



```
1. class Vehicle{
2.   color;
3.   type; //Electric or petrol or diesel based
4.   no_of_tyres; // 4 wheelers or 2 wheelers
5.   constructor(color, type,no_of_tyres){
6.     this.color = color;
7.     this.type = type;
8.     this.no_of_tyres = no_of_tyres;
9.   };
10. };
```

As seen above, every class is defined using the "class" keyword followed by classname. Let's see step by step what is happening here:
In line 1, we defined Vehicle class with the help of "class" keyword.
From lines 2 – 4, we have defined a few properties of the Vehicle class.

## Encapsulation:

Wrapping or binding of data and functions in a single unit is called encapsulation.

If other classes need to access the data, then they need to make use of the getter or setter method.

Here the data is hidden and this is done by declaring properties as private.

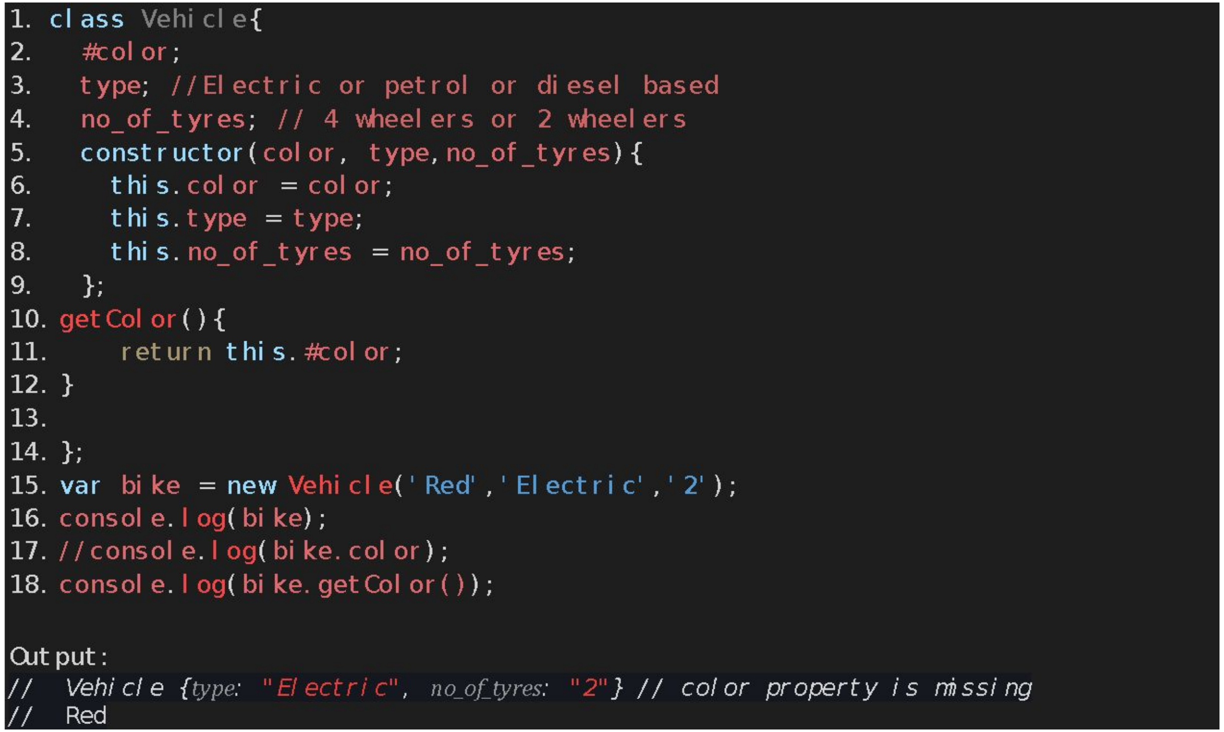
By default, every property of a class is **public.** This means it can be directly accessed outside its class and can be misused.

To prevent this we make these **private**.

Private properties cannot be accessed outside its class.

To make a property as private we use “**#**” symbol in Javascript.

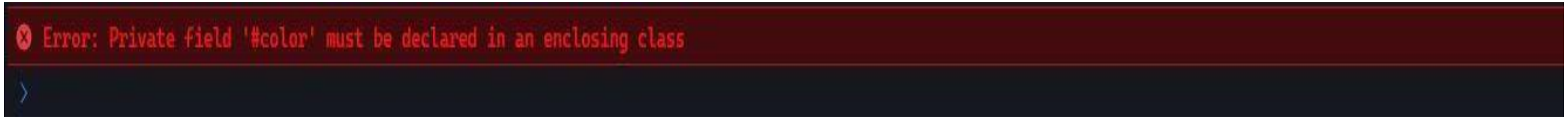
## Example:



```
class Vehicle{
   #color;
   type; //Electric or petrol or diesel based
   no_of_tyres; // 4 wheelers or 2 wheelers
   constructor(color, type,no_of_tyres){
      this.color = color;
      this.type = type;
      this.no_of_tyres = no_of_tyres;
   };
getColor(){
      return this.#color;
}

};
var bike = new Vehicle('Red','Electric','2');
console.log(bike);
//console.log(bike.color);
console.log(bike.getColor());


Output:
//  Vehicle {type: "Electric", no_of_tyres: "2"} // color property is missing
//  Red
```

## Example continue:

- If you compare this example with the previous example you will notice the following changes,
- At line 2, we have used **“#”** symbol before “color” property. When some method or property is prefixed with “#” symbol, it makes the method or property as private member and you won’t be directly able to access the value outside the class as seen in line 16 console output.
- At line 10, we have defined a getter method named as “getColor” to fetch the value of color at line 18.
- If you uncomment line 17, you will be able to see the below error,



```
Error: Private field '#color' must be declared in an enclosing class
>
```

## Abstraction:

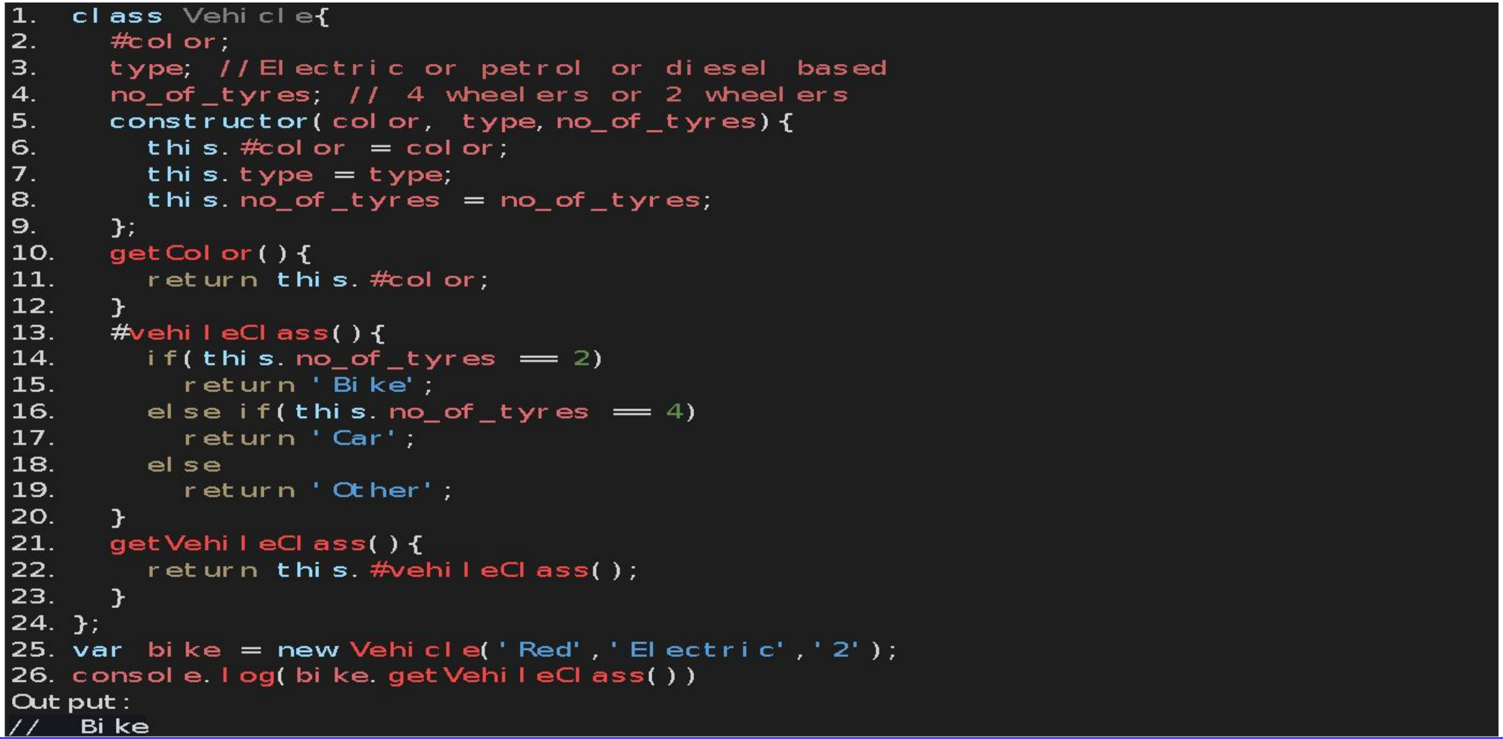
Abstraction is a process of showing the essential details and hiding the non-essential details.

It helps in avoiding code redundancy

Helps in increasing the security of the application.

Here we hide the implementation part of the application.

## Example:



```
class Vehicle{
  #color;
  type; //Electric or petrol or diesel based
  no_of_tyres; // 4 wheelers or 2 wheelers
  constructor(color, type,no_of_tyres){
    this.#color = color;
    this.type = type;
    this.no_of_tyres = no_of_tyres;
  };
  getColor(){
    return this.#color;
  }
  #vehileClass(){
    if(this.no_of_tyres == 2)
      return 'Bike';
    else if(this.no_of_tyres == 4)
      return 'Car';
    else
      return 'Other';
  }
  getVehileClass(){
    return this.#vehileClass();
  }
};
var bike = new Vehicle('Red','Electric','2');
console.log(bike.getVehileClass())
Output:
//   Bike
```

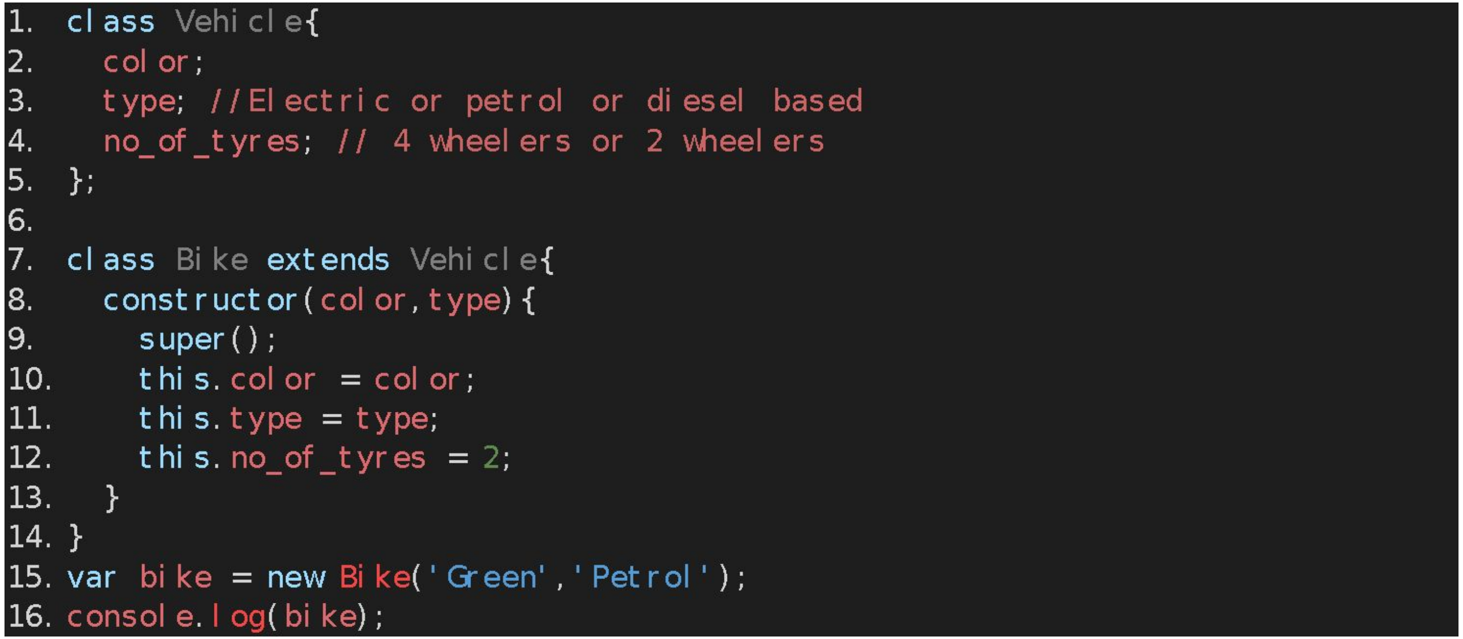
## Example continue:

- In the above program, you can see at line 21, a method is added “getVehicleClass”.
- The user is not concerned of what’s going on inside getVehicleClass method. User just calls the method at line 23 and gets the output.

## Inheritance:

- When a class inherits the properties and methods of another class, this is known as inheritance.
- The class from which the properties and methods are inherited is known as **“Parent class”.**
- The class that inherits these properties and methods is known as **“Child class”.**
- Inheritance helps in code reusability.
- **“extends and super”** keywords are used to establish parent-child relationships.

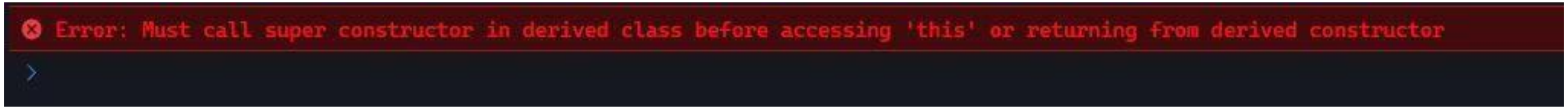
## Example:



```
class Vehicle{
  color;
  type; //Electric or petrol or diesel based
  no_of_tyres; // 4 wheelers or 2 wheelers
};

class Bike extends Vehicle{
  constructor(color,type){
    super();
    this.color = color;
    this.type = type;
    this.no_of_tyres = 2;
  }
}
var bike = new Bike('Green','Petrol');
console.log(bike);
```

## Example continue:

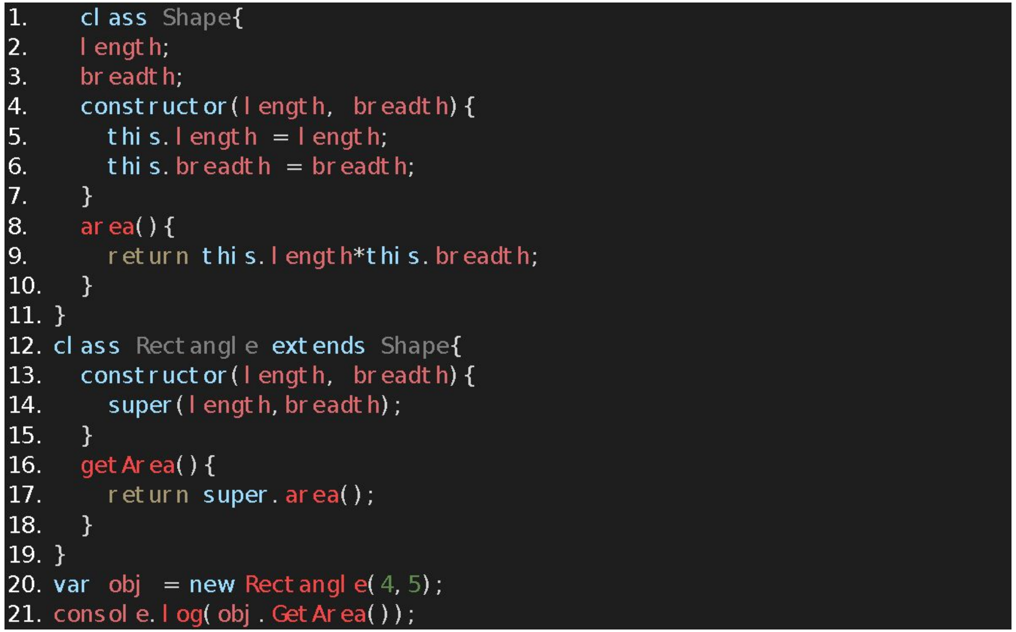
- Here we can see we have created two classes Vehicle and Bike.
- Bike is inheriting the properties of Vehicle class by extending it at line 7.
- Inside constructor, we can see we have used “**super()**”. The work of “super()” is to access and call method of the object’s parent. In constructor, the **super** keyword must be used before **“this”** keyword is used.
- If you comment line 9, you will get the below mentioned error,



```
Error: Must call super constructor in derived class before accessing 'this' or returning from derived constructor
>
```

## Super():

- Super is a keyword which is used to access and call methods of the parent class.
- To access the constructor of the parent class we use **super([arguments]).** Here arguments is an optional field and depends on the parent constructor.
- To access the method and parameters of the parent class we make use of the **dot(.)operator** suffixed after the super keyword.
- Let’s see an example of parameterised parent constructor using super().



```
1.     class Shape{
2.     length;
3.     breadth;
4.     constructor(length, breadth){
5.       this.length = length;
6.       this.breadth = breadth;
7.     }
8.     area(){
9.       return this.length*this.breadth;
10.    }
11. }
12. class Rectangle extends Shape{
13.    constructor(length, breadth){
14.      super(length,breadth);
15.    }
16.    getArea(){
17.      return super.area();
18.    }
19. }
20. var obj = new Rectangle(4,5);
21. console.log(obj.GetArea());
```

In the above example we can see at line 14, how we have passed parameter to Shape class from the Rectangle class.
Notice how we are using dot(.) operator at line 17, to get the area calculated in parent class.

## Polymorphism:

Polymorphism is defined as a way to perform one operations or action in multiple forms.

There are two types of polymorphism:

1. Compile-time polymorphism (Method Overloading)
2. Run-time polymorphism (Method Overriding)

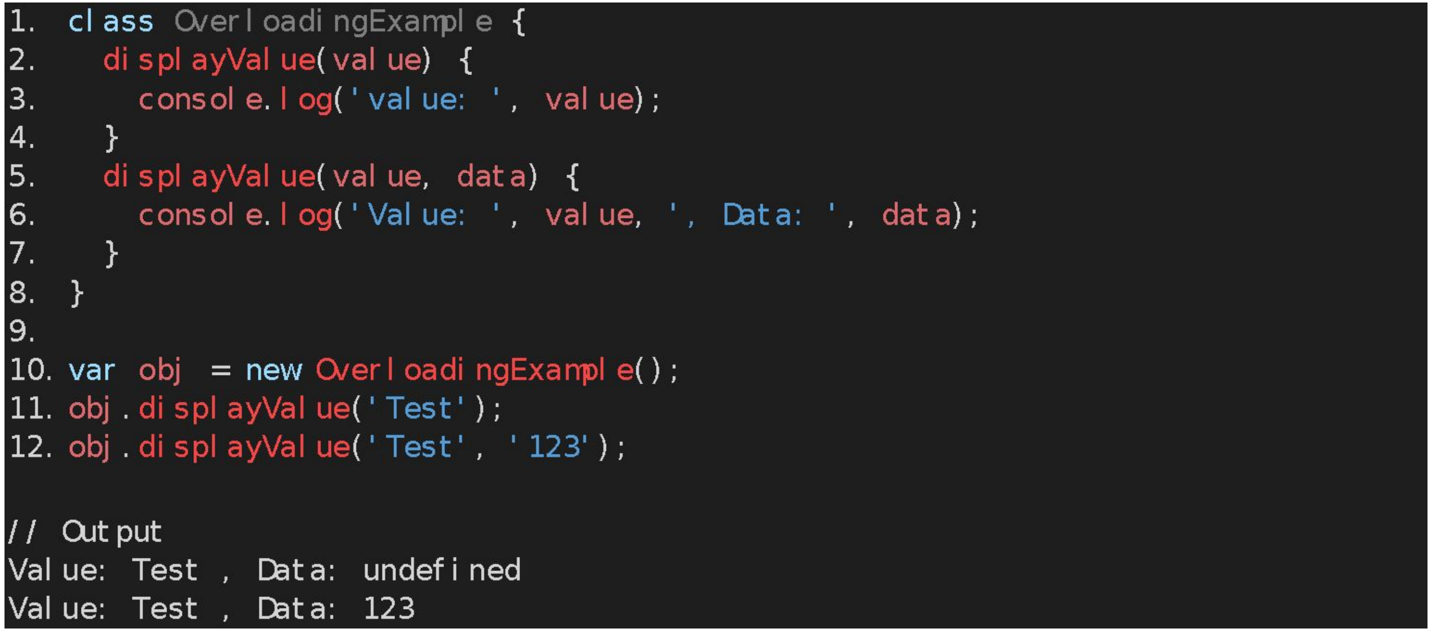
## Method-Overloading:

It is implemented by providing the same method name but with different parameter lists or parameter types.

Javascript does not support method overloading as its not type-safe.

If we try to implement overloading, it will just override the previous method with former parameters.

## Example:



```
class OverloadingExample {
   displayValue(value) {
      console.log('value: ', value);
   }
   displayValue(value, data) {
      console.log('Value: ', value, ', Data: ', data);
   }
}

var obj = new OverloadingExample();
obj.displayValue('Test');
obj.displayValue('Test', '123');

// Output
Value: Test , Data: undefined
Value: Test , Data: 123
```

In the above example,

At line 2, we created a method displayValue() with one parameter and we have defined same method again at line 5, but with two parameters.

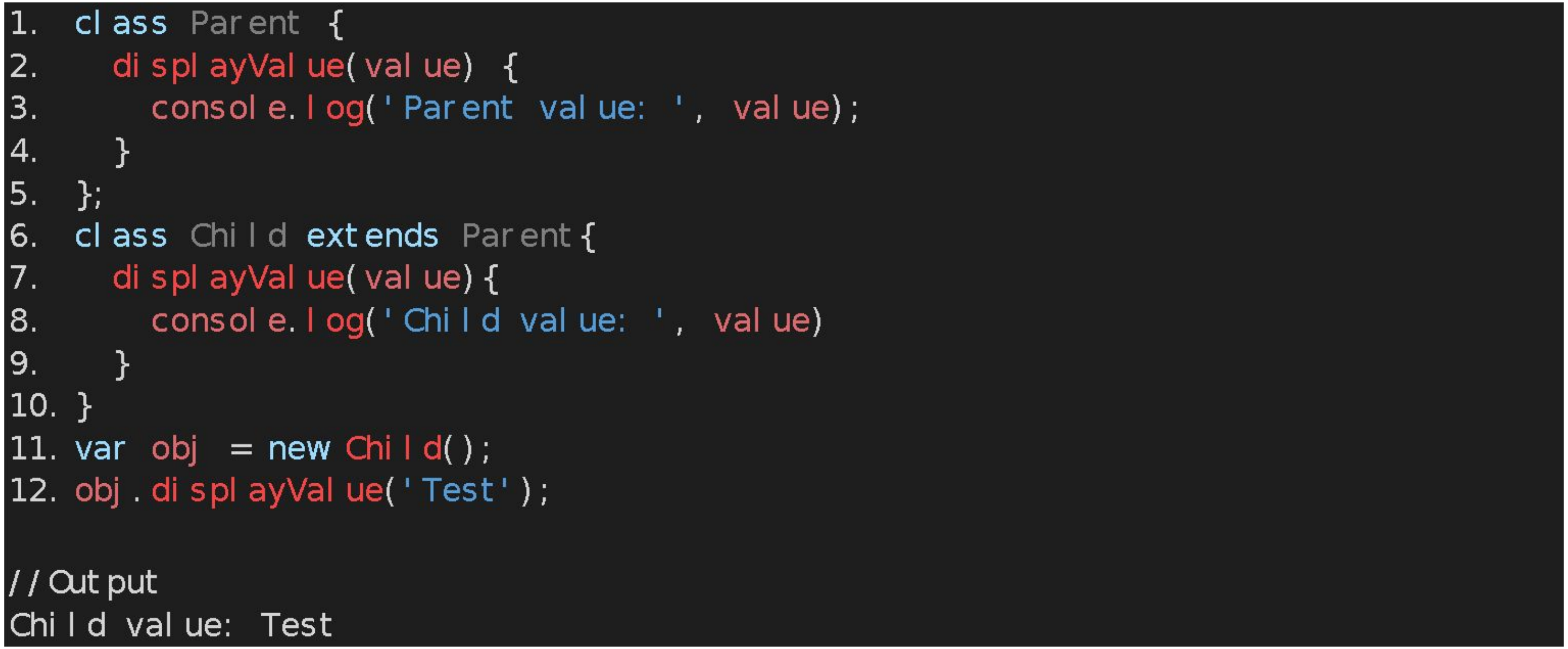
When we try to call displayValue at line 11, ideally it should call method at line 2, but as javascript doesn't support method overloading its just overriding the method at line 2 with line 5 and we get the output as shown above.

## Method-Overriding:

It is implemented by providing the same method name, parameters but with different body signature.

This can be achieved using inheritance only.

## Example:



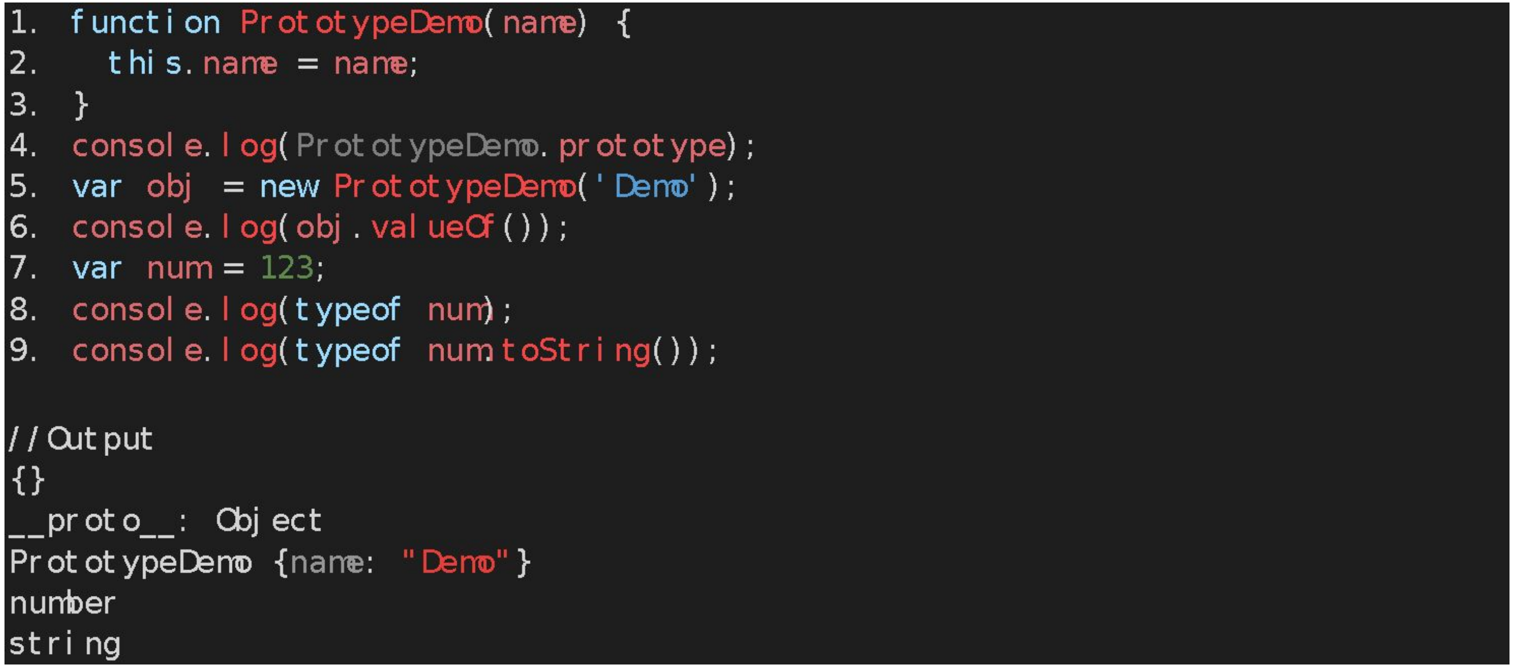
```
class Parent {
  displayValue(value) {
    console.log('Parent value: ', value);
  }
};
class Child extends Parent{
  displayValue(value){
    console.log('Child value: ', value)
  }
}
var obj = new Child();
obj.displayValue('Test');

//Output
Child value: Test
```

- Here we can see **displayValue** method is present in both Parent and Child classes.
- The method name and parameter are same and only body definition is different.
- So, this is called method overriding.
- If you comment lines from 7-9, you will still be able to access displayValue method but this time you will be accessing Parent class method instead of Child class.

## Prototypes:

- Prototype is defined as a way by which an object in Javascript, inherits existing properties and methods of another object.
- Prototype allows us to add new properties and methods to the object constructor.
- Prototype is a global property which is available with almost all the objects.
- Javascript automatically adds prototype property inside a function as soon as we create it.
- Prototype has some pre-defined functions like toString(), toLocaleString(), valueOf(),etc. More can be seen in Prototype Chaining section.

## Example:



```
function PrototypeDemo(name) {
  this.name = name;
}
console.log(PrototypeDemo.prototype);
var obj = new PrototypeDemo('Demo');
console.log(obj.valueOf());
var num = 123;
console.log(typeof num);
console.log(typeof num.toString());

//Output
{}
__proto__: Object
PrototypeDemo {name: "Demo"}
number
string
```

- In the above example, we created a PrototypeDemo function and when we try to console the output, a __proto__ object is added automatically.
- We can see at line 6, valueOf() is used. Even though we haven't defined it in PrototypeDemo function. So how come its not throwing any error. This is because it is coming from Prototype function as discussed earlier.
- Similarly, at line 9, we had used toString() on a variable "num".
- This is the prototype property that we discussed in the above definition.

## Prototype Classes:

- Javascript is an object-based language based on prototypes.
- Before ECMAScript 2015 ( also known as ES6 ), we used to define a class using functions as shown in the example below.
- Javascript classes, introduced in ES6 are primarily syntactic sugar over JavaScript's existing prototype-based inheritance.

## Example

```javascript
function PrototypeDemo(name){
   this.name = name;
}
var obj = new PrototypeDemo("ProtoType Demo")
console.log(typeof(obj)); //Output will be “Object”.
```

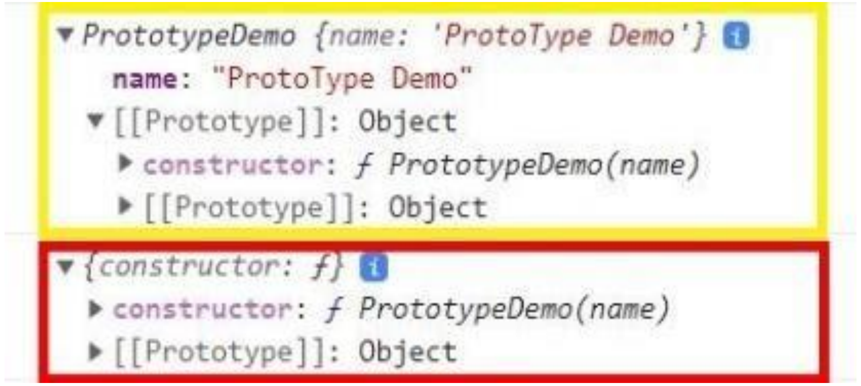
- In this example we can see how we used to implement classes before ES6. We don’t have class keyword as we have seen before. It's just a simple function, as Javascript is object based so when we console the “obj " we will get the type as “Object”.

## Prototype Chaining:

- Every Javascript object has a pre-defined property known as prototype.
- This prototype is nothing but an object itself. As it is an object, it will contain its own prototype.
- This will create a chain of prototypes inside another prototype known as Prototype Chaining.
- The chain will end as soon as we reach a prototype with **null** value.
- **Object.getPrototypeOf()** method is used to fetch the prototype object.

## Example:

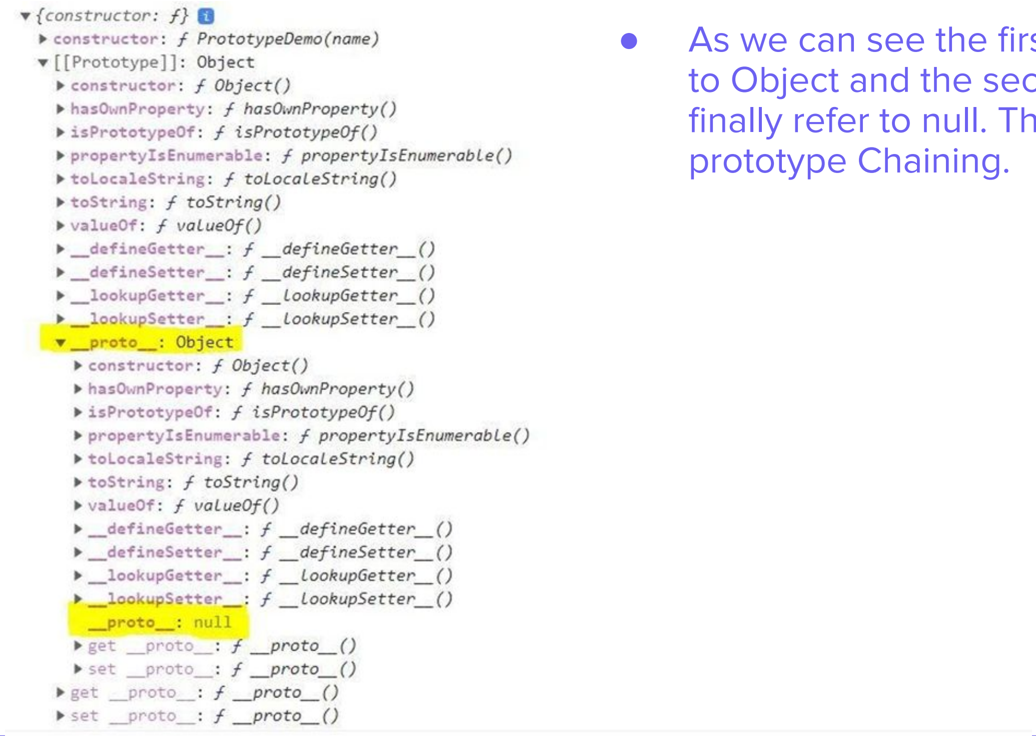
```
function PrototypeDemo(name){
   this.name = name;
}
var obj = new PrototypeDemo("ProtoType Demo");
console.log(obj);
console.log(Object.getPrototypeOf(obj));
```



```
▼ PrototypeDemo {name: 'ProtoType Demo'} ℹ
    name: "ProtoType Demo"
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ PrototypeDemo(name)
    ▶ [[Prototype]]: Object
▼ {constructor: ƒ} ℹ
  ▶ constructor: ƒ PrototypeDemo(name)
  ▶ [[Prototype]]: Object
```

- In the above output, we can see that by using Object.getPrototypeOf() we can get the prototype information as highlighted inside red box.
- Now if we expend [[Prototype]]: Object , we will be able to see __proto__.

## Example continue:



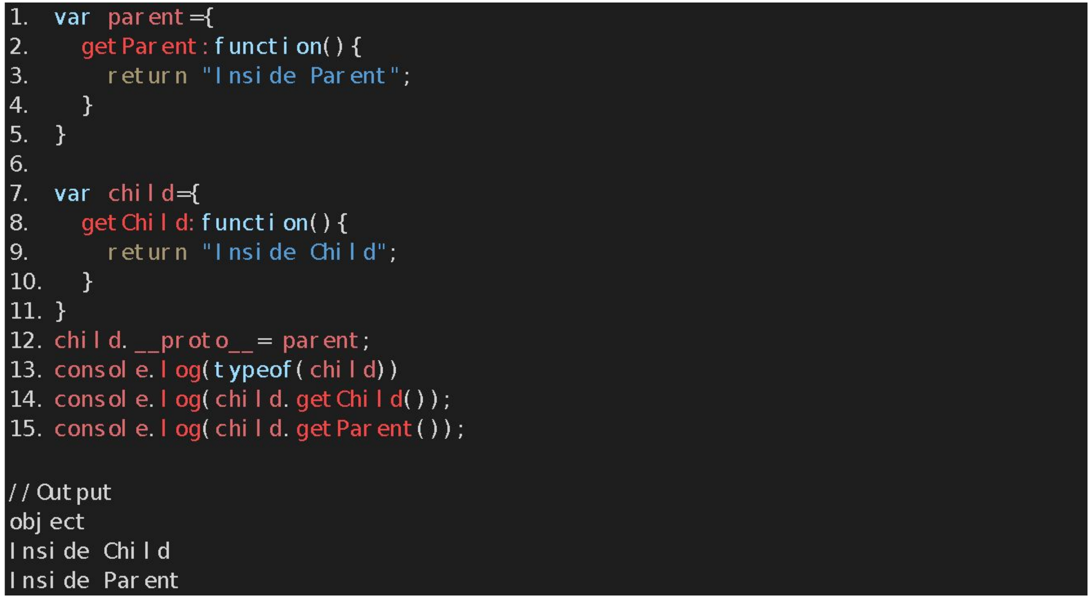
```
▼{constructor: ƒ}
  ▶constructor: ƒ PrototypeDemo(name)
  ▼[[Prototype]]: Object
    ▶constructor: ƒ Object()
    ▶hasOwnProperty: ƒ hasOwnProperty()
    ▶isPrototypeOf: ƒ isPrototypeOf()
    ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶toLocaleString: ƒ toLocaleString()
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
    ▶__defineGetter__: ƒ __defineGetter__()
    ▶__defineSetter__: ƒ __defineSetter__()
    ▶__lookupGetter__: ƒ __lookupGetter__()
    ▶__lookupSetter__: ƒ __lookupSetter__()
    ▼__proto__: Object
      ▶constructor: ƒ Object()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
       __proto__: null
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
    ▶get __proto__: ƒ __proto__()
    ▶set __proto__: ƒ __proto__()
```

- As we can see the first __proto__ refer to Object and the second __proto__ finally refer to null. This is known as prototype Chaining.

## Prototypal Inheritance:

- In order to implement prototypal inheritance, we make use of prototype chaining.
- Any function can be added to an object in the form of property.
- Every object in javascript has prototype property which is either null or reference to another object.
- When we try to read a property from child object and if its null, then it will read the property from parent object, if defined using "__proto__" keyword to create a parent-child relationship.

## Example:



```
1.  var parent={
2.    getParent:function(){
3.      return "Inside Parent";
4.    }
5.  }
6.
7.  var child={
8.    getChild:function(){
9.      return "Inside Child";
10.   }
11. }
12. child.__proto__= parent;
13. console.log(typeof(child))
14. console.log(child.getChild());
15. console.log(child.getParent());

//Output
object
Inside Child
Inside Parent
```

Above we created a relationship between child and parent using __proto__ at line12.
And we are able to access parent method at line 15.

## MCQ:

1. Predict the output:

```
class Vehicle{
 color;
 type; //Electric or petrol or diesel based
 no_of_tyres; // 4 wheelers or 2 wheelers
  constructor(color, type,no_of_tyres){
   this.color = color;
   this.type = type;
   this.no_of_tyres = no_of_tyres;
 };
};
var bike = new Vehicle(Green,'Electric','2');
console.log(bike.color);
```

1. **Output:** Green
2. undefined
3. Error: Green is not defined
4. None of the above

## MCQ:

1. Predict the output:

```
class Vehicle{
 color;
 type; //Electric or petrol or diesel based
 no_of_tyres; // 4 wheelers or 2 wheelers
  constructor(color, type,no_of_tyres){
   this.color = color;
   this.type = type;
   this.no_of_tyres = no_of_tyres;
 };
};
var bike = new Vehicle(Green,'Electric','2');
console.log(bike.color);
```

1. **Output:** Green
2. undefined
3. Error: Green is not defined
4. None of the above

- **Answer:** Error: Green is not defined

## MCQ:

2. Predict the output:

```
class Vehicle{
 color;
 type; //Electric or petrol or diesel based
 no_of_tyres; // 4 wheelers or 2 wheelers
  constructor(color, type,no_of_tyres){
   this.color = color;
   this.type = type;
   this.no_of_tyres = no_of_tyres;
 };
};
var bike = new
Vehicle("Green",'Electric','2');
onsole.log(bike.color);
bike = new Vehicle();
console.log(bike.color);
```

1. **Output:** Green
2. Green, Green
3. Green, undefined
4. None of the above

# MCQ:

2. Predict the output:

```
class Vehicle{
 color;
 type; //Electric or petrol or diesel based
 no_of_tyres; // 4 wheelers or 2 wheelers
  constructor(color, type,no_of_tyres){
   this.color = color;
   this.type = type;
   this.no_of_tyres = no_of_tyres;
 };
};
var bike = new
Vehicle("Green",'Electric','2');
onsole.log(bike.color);
bike = new Vehicle();
console.log(bike.color);
```

1. **Output:** Green
2. Green, Green
3. Green, undefined
4. None of the above

- **Answer:** Green, undefined

## MCQ:

3. Predict the output:

```
class Parent {
   displayValueP(value) {
        console.log('Parent value: ', value);
   }
};
class Child extends Parent{
   displayValue(value){
        console.log('Child value: ', value)
      }
}
var obj = new Child();
obj.displayValueP('Test’);
```

Output:

1. Child value: Test
2. Parent value: Test
3. Error
4. None of the above

## MCQ:

3. Predict the output:

```
class Parent {
   displayValueP(value) {
        console.log('Parent value: ', value);
   }
};
class Child extends Parent{
   displayValue(value){
        console.log('Child value: ', value)
      }
}
var obj = new Child();
obj.displayValueP('Test’);
```

Output:

1. Child value: Test
2. Parent value: Test
3. Error
4. None of the above

- **Answer:** Parent value: Test

## MCQ:

4. Can we use “this” keyword in the child class constructor without using “super” keyword?
    a. Yes
    b. No

## MCQ:

4. Can we use “this” keyword in the child class constructor without using “super” keyword?
    a. Yes
    b. No

- **Answer:** No

## Practice Questions

1. Write a program to define a class “Shape”. This class will have the properties length, breadth and height. Create a parameterised constructor that takes input as length, breadth and height. Define a method area() inside the class and it should return value as length*breadth*height.Create an Object of shape class and call the area() method to fetch the area.

2. Write a program to define a class “Shape”. This class will have the private properties length, breadth and height. Create setter and getter methods for all the private properties. Define a method area() inside the class and it should return value as length*breadth*height.Create an Object of shape class and call the area() method to fetch the area.

3. Write a program that contains Cube class and Circle class. These classes should inherit the Shape class.  Set the properties value if required using constructor and override the area() method (if required) to return the area of Cube and Circle class.

## Next Class teaser:

In the next class we will cover the following concepts:

- Intro to Asynchronicity
- Threads and Processes
- Callbacks
- Call Stack
- Timer API

## THANK YOU