# Prototypal Chaining

This lesson teaches the concept of prototype chaining in detai by using an example.

**We'll cover the following**



In the previous lesson, the `[[Prototype]]` property of objects was discussed.

We discussed the following code:

```
//Shape object
var Shape={
  name: 'Rectangle',
  sides: 4
}

//Rectangle object
var Rectangle = {
  length: 3,
  width: 5
}

//setting [[Prototype]] of Rec
Rectangle.__proto__ = Shape

//creating an object instance
console.log("Name of shape is:
console.log("Number of sides a
console.log("Length is:",Recta
console.log("Width is:",Rectan
```

Let's delve into the details of the code above.

When the *prototype* property of `Rectangle` is set to `Shape`, it is able to access

all the properties present in `Shape`. So, upon accessing, if a property is not found in the object, such as if the `name` property is not found in `Rectangle`, JavaScript will automatically take it from the prototype of the object, `Shape`. This is known as **prototypal inheritance**.

```
//Shape object
var Shape={
  name: 'Rectangle',
  sides: 4
}
//Rectangle object
var Rectangle = {
  length: 3,
  width: 5
}
 //setting [[Prototype]] of Rectangle equal to Shape
Rectangle.__proto__ = Shape
console.log(Rectangle.__proto__)
console.log(Shape.__proto__)
```

As seen from the code above, since the prototype of `Rectangle` points to `Shape`, upon accessing its prototype, it displays the properties present in `Shape`. Since `Shape` does not have its prototype set to another object, it will point to null. Hence, it will display nothing upon access.

## What is Prototype Chaining? #

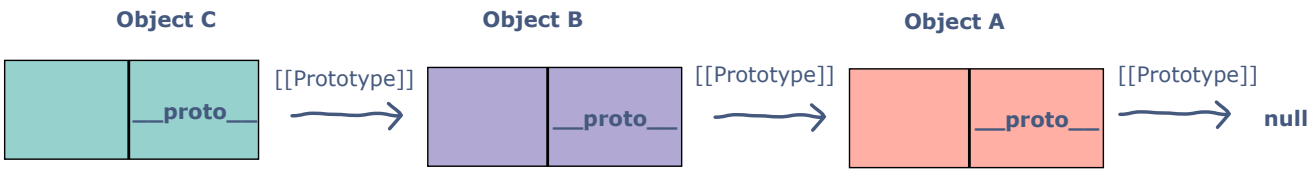
Prototypal inheritance uses the concept of **prototype chaining**. Let's learn what it is.

As discussed, every object created contains a private property called `[[Prototype]]` which points either to another object or `null`. Let's consider an example where there is an object **C** whose `[[Prototype]]` property points to object **B**. Object B's `[[Prototype]]` property, in turn, points to object **A**. This could go on and on resulting in a chain known as the **prototype chain**.

The *prototype chain* is used in prototypal inheritance. Whenever a property is to be found in an object, it is first searched for in the object itself; if not found, it is then searched for on that object's prototype. If it is still not found, it gets searched for in the object's prototype's prototype. Thus, the entire prototype

chain gets traversed for the property to be found until `null` is reached.

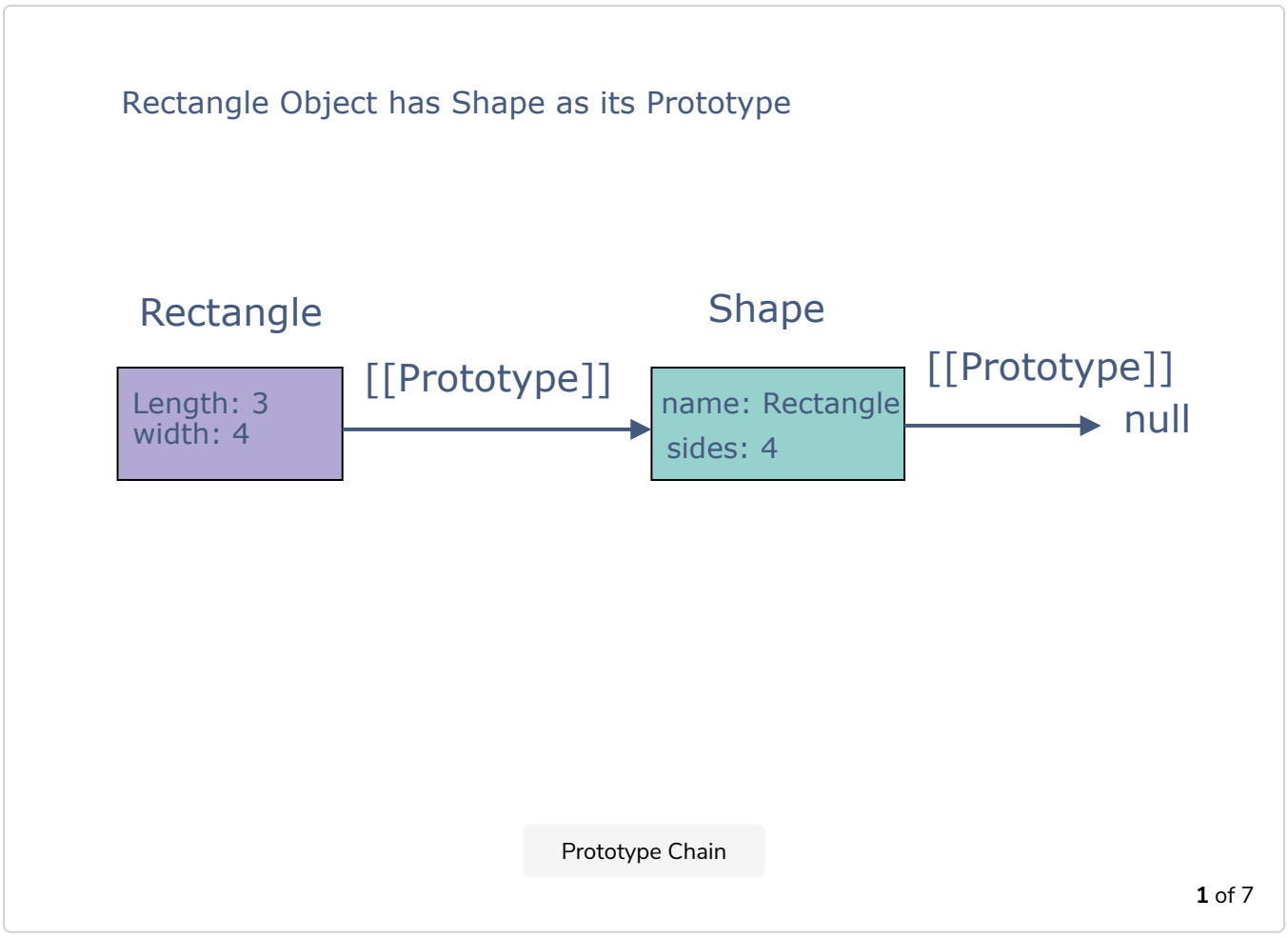
Take a look at the illustration below in order to understand this concept better:




## Applying Prototype Chaining #

How is prototype chaining used in our example?

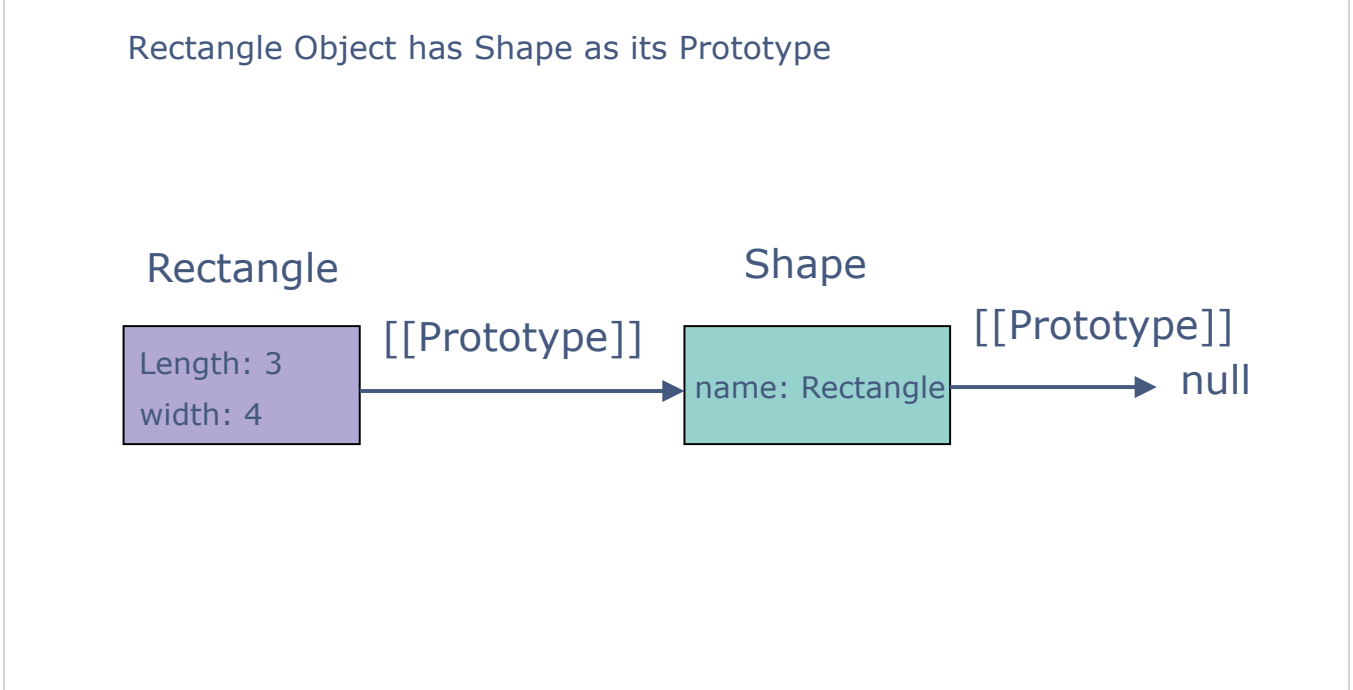
In **line 14**, `Shape` is set to be the prototype of `Rectangle`. So in **lines 17** and **18** when the `name` and `sides` properties are called from `Rectangle`, they are first searched for in its object. When they aren't found, JavaScript then searches for them in its `[[Prototype]]`, `Shape` in our case, from where it takes both properties since they are present in it. These properties are referred to as **inherited properties**.

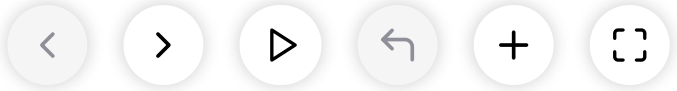


If the `name` and `sides` properties weren't found in `Shape`, the prototype of `Shape` would have been searched next, which would have pointed to `null`. Since `null` doesn't have a prototype, that would mean the end of the chain had been reached hence `undefined` would've been returned.

Let's take a look at an example of that:

```
//Shape object
var Shape={
  name: 'Rectangle',
  //sides property removed from Shape
}
//Rectangle object
var Rectangle = {
  length: 3,
  width: 5
}
 //setting [[Prototype]] of Rectangle equal to Shape
Rectangle.__proto__ = Shape
//accessing the sides property from Rectangle
console.log(Rectangle.sides)
```





Now that you know how prototypal inheritance uses prototype chaining in objects, let's discuss the same concept for when objects are made through *constructor functions* in the next lesson.