4

Object Oriented JavaScript

JavaScript's most fundamental data type is the Object data type. JavaScript objects can be seen as mutable key-value based collections. In JavaScript, arrays, functions and RegEx are objects while numbers, strings and booleans are object-like constructs that are immutable but have methods.

Understanding objects

Before we start looking at how JavaScript treats objects, we should spend some time on object oriented paradigm. Like most programming paradigms, Object oriented programming also emerged from the need for managing complexity. The main idea is to divide the entire system onto smaller pieces that are isolated from each other. If these small pieces can hide as much implementation details as possible, they become easy to use. A classic car analogy will help you understand this very important point about OOP. When you drive a car, you operate on the interface – the steering, the clutch, break and the accelerator. Your view of using the car is limited by this interface – which makes it possible for us to drive the car. This interface is essentially hiding all the complex systems that really drive the car, such as the internal functioning of its engine, its electronic system, etc. As a driver you don't bother about those complexities. Similar idea is the primary driver of OOP. An object hides the complexities of how to implement a particular functionality and exposes a limited interface to the outside world. All other systems can use that interface without really bothering about the internal complexity hidden from view. Additionally, usually an object hides its internal state from other objects and prevents its direct modification. This is an important aspect of OOP. In a large system where a lot of objects call other object's interfaces, if you allow them to modify the internal state of such objects, things can go really bad. OOP operates on the

Formatted: Left: 3,81 cm, Right: 3,81 cm, Top: 4,14 cm, Bottom: 4,77 cm, Width: 21,59 cm, Height: 27,94 cm, Header distance from edge: 3,48 cm, Footer distance from edge: 4,14 cm, Different first page header

Deleted: '

Deleted: are

Deleted: object

Comment [MP1]: The introductory section is too long. It would be better to have a short section mentioning what we would be covering in this chapter.

Also, it would be better to have heading level 1 as Understanding Javascript objects OR
Understanding objects

Formatted: Heading 1;Heading 1 [PACKT]

Deleted: accelarator

Deleted: '

Deleted: the

Deleted: of an object

Deleted: '

idea that the state of an object is inherently hidden from the outside world and it can be changed only via controlled interface operations.

OOP was an important idea and a definite step forward from the traditional structured programming. However, many feel that OOP is overdone. Most OOP systems define complex and unnecessary class and type hierarchies. Another big drawback was that in pursuit of hiding the state, OOP considered the object state almost immaterial. Though hugely popular, OOP was clearly flawed on many areas. Still, OOP did have some very good ideas, especially hiding the complexity and exposing only the interface to the outside world. JavaScript picked up a few good ideas and built its object model around those. Luckily, that makes JavaScript objects very versatile. In their seminal work *Design Patterns: Elements of Reusable Object Oriented Software,* the Gang of Four gave two fundamental principles of better object oriented design:

- Program to an interface and not to an implementation
- Object composition over class inheritance

These two ideas are really against how classical OOP operates. Classical operates on inheritance which exposes parent classes to all child classes. Classical inheritance hence tightly couples children to its parents. Classical inheritance also has a limitation where you can only enhance the child class within limit of the parent classes. You can't fundamentally differ from what you have got from the ancestors. This inhibits reuse. Classical inheritance has several other problems.

- Inheritance introduces tight coupling. Child classes have knowledge about their ancestors. This tightly couples a child class with its parent.
- When you subclass from a parent, you don't have a choice to select what you want to inherit and what you don't. Joe Armstrong (inventor of Erlang) explain this situation very well – his now famous quote : *The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.*

Behaviour of JavaScript objects

With this background with us, let's explore how JavaScript objects behave. In broad terms, an object contains properties – defined as key-value pair. A property key (name) can be a string and the value can be any valid JavaScript value. You can create objects using object literals. The following snippet shows how object literals are created:

Deleted: inheretently

Deleted: principals

Deleted: compositon

Comment [IM2]: •Here you have to explain also what "Program to an interface and not to an implementation" concretely means.

Deleted: '

Deleted: '

Deleted: '

Deleted: '

Formatted: Heading 2;Heading 2 [PACKT]

```
var nothing = {};
var author = {
    "firstname": "Douglas",
    "lastname": "Crockford"
}
```

A property's name can be any string or an empty string. You can omit quotes around the property name if the name is a legal JavaScript name. So quotes are required around "first- name", but are optional around firstname. Commas are used to separate the pairs. You can nest objects like the following:

```
var author = {
  firstname : "Douglas",
  lastname : "Crockford",
  book : {
    title:"JavaScript- The Good Parts",
    pages:"172"
  }
};
```

Properties of an object can be accessed by using two notations: the array-like notation and the dot notation. According to the array-like notation, you can retrieve the value from an object by wrapping a string expression in a []. If the expression is a valid JavaScript name, you can use the dot notation by using the '.' instead. Using a '.' is a preferred method of retrieving values from an object.

```
console.log(author['firstname']); //Douglas
console.log(author.lastname);     //Crockford
console.log(author.book.title);   // JavaScript- The Good Parts
```

You will get an undefined if you attempt to retrieve a non-existent value. The following would return undefined.

```
console.log(author.age);
```

A useful trick is to use '||' operator to fill in default values in this case.

```
console.log(author.age || "No Age Found");
```

You can update values of an object by assigning the new value to the property.

```
author.book.pages = 190;
```

Deleted: “
Deleted: ”
Deleted: “
Deleted: ”
Deleted: “
Deleted: ”
Deleted: “
Deleted: ”
Deleted: ’
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: You
Deleted: a
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: '
Deleted: '
Deleted: ‘
Deleted: ’
Deleted: "
Deleted: "

```
console.log(author.book.pages); //190
```

If you observe closely, you will realize that the object literal syntax you see above is very similar to JSON format.

Methods are properties of an object that can hold function values. For example:

```
var meetingRoom = {};
meetingRoom.book = function(roomId){
  console.log("booked meeting room -"+roomId);
}
meetingRoom.book("VL");
```

Prototypes

Apart from the properties that we add to an object, there is one default property for almost all objects called a **prototype**. When an object does not have a requested property, JavaScript goes to its prototype to look for it. Function Object.getPrototypeOf () returns the prototype of an object.

Many programmers consider prototypes closely related to objects inheritance, they are indeed a way of defining object types, but fundamentally they are closely associated with functions.

Prototypes are used as a way to define properties and functions that will be applied to instances of objects. The prototype's properties eventually become properties of the instantiated objects. Prototypes can be seen as blueprints for object creation. They can be seen as analogous to *classes* in object oriented languages. Prototypes in JavaScript are used to write a classical style object-oriented code and mimick classical inheritance. Let's revisit our earlier example:

```
var author = {};
author.firstname = 'Douglas';
author.lastname = 'Crockford';
```

In the code snippet above, we are creating an empty object and we are assigning individual properties. You will soon realize that this is not a very standard way of building objects. If you know object oriented programming already, you will immediately see that there is no encapsulation and usual class structure. JavaScript provides a way around this. You can use 'new' operator to instantiate an object via constructors. However, there is no no concept of a class in JavaScript, and it is important to note that

Comment [IM3]: Here you can introduce the parse and stringify methods of the JSON object. See here: https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/JSON

Deleted: .

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Formatted: Heading 2;Heading 2 [PACKT]

Deleted: With this basic understanding in place, I think we are prepared to look at prototypes.

Formatted: Font:Bold

Deleted: they are actually a

Deleted: feature of

Deleted: means of defining

Deleted: functionality

Deleted: ’

Deleted: y

Deleted: become

Deleted: s

Deleted: and hence serves as a

Comment [VA4]: Very generic statement, not sure if this can be modified.

Deleted: Most popular use of p

Deleted: is in

Deleted:

Deleted: producing

Deleted: of

Deleted: ’

Deleted: '

Deleted: '

Deleted: '

Deleted: '

Deleted: ‘

Deleted: ’

Comment [VA5]: Not sure how to modify this very generic statement

Deleted: but as there is

the 'new' operator is applied to the constructor function. To clearly understand this, let's look at the following example:

```
//A function that returns nothing and creates nothing
function Player() {}

//Add a function to the prototype property of the function
Player.prototype.usesBat = function() {
  return true;
}

//We call player() as a function and prove that nothing happens
var crazyBob = Player();
if(crazyBob === undefined){
  console.log("CrazyBob is not defined");
}

//Now we call player() as a constructor along with 'new'
//1. The instance is created
//2. method usesBat() is derived from the prototype of the
function
var swingJay = new Player();
if(swingJay && swingJay.usesBat && swingJay.usesBat()){
  console.log("SwingJay exists and can use bat");
}
```

In the example above, we have a function player() that does nothing. We invoke it in two different ways The first call of the function is as a normal function and second call is as a constructor – please note the use of new() operator in this call. Once the function is defined, we add a method to usesBat() to it. When this function is called as a normal function, the object is not instantiated and we see 'undefined' assigned to 'crazyBob'. However, when we call this function with 'new' operator, we get a fully instantiated object 'swingJay'.

Deleted: the
Deleted: ‘
Deleted: ’
Deleted: in fact
Deleted: ’
Comment [MM6]: To the reviewers:
Do you feel anything more needs to be added to this? Or do you think anything needs to be modified?
Deleted: "
Deleted: "
Deleted: '
Deleted: '
Deleted: "
Deleted: "
Deleted: ,
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’

Instance properties versus Prototype Properties

Instance properties are the properties, which are part of the object instance itself. For example:

```
function Player() {
  this.isAvailable = function() {
    return "Instance method says - he is hired";
  };
}
Player.prototype.isAvailable = function() {
  return "Prototype method says - he is Not hired";
};
var crazyBob = new Player();
console.log(crazyBob.isAvailable());
```

When you run this example, you will see that "Instance method says - he is hired" is printed. The function isAvailable() defined inside function Player() is called an instance of Player. This means apart from attaching properties via the prototype, you can use 'this' keyword to initialize properties within a constructor. When we have same functions defined as an instance property and also as a prototype, instance property takes precedence. The rules governing the precedence of the initialization are as follows:

- Properties are tied to the object instance from the prototype.
- Properties are tied to the object instance within the constructor function.

This example brings us to the use of `this` keyword. It is easy to get confused by 'this' keyword because `this` keyword behaves differently in JavaScript. In other OO languages like Java, the `this` keyword refers to the current instance of the class. In JavaScript the value of this is determined the invocation context of function and where it is called. Let's understand how this behaviour needs to be carefully understood.

- When 'this' is used in global context – When 'this' is called in global context, it is bound to the global context. For example, in case of a browser, the global context is usually 'window'. This is true for functions also. If you use 'this' inside a function which is defined in global context, it is still bound to the global context because the function is part of the global context.

```
function globalAlias(){
```

Formatted: Heading 1;Heading 1 [PACKT]
Deleted: /s
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: that in addition to
Deleted: we can initialize values within the constructor function via this keyword
Deleted: .
Deleted: operations is important and goes as follows
Formatted: Bulleted + Level: 1 + Aligned at: 0,63 cm + Tab after: 1,27 cm + Indent at: 1,27 cm
Deleted: bound
Deleted:
Deleted: added
Deleted:
Deleted: ‘
Formatted: Font:Lucida Console, 9,5 pt, Font color: R,G,B (116,121,89)
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Formatted: Font:Lucida Console, 9,5 pt, Font color: R,G,B (116,121,89), English (US)
Deleted: ’
Formatted ... [1]
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’

```
  return this;
}
console.log(globalAlias()); //[object Window]
```

- When 'this' is used inside object method – in this case, 'this' is assigned or bound to the enclosing object. Please note that the enclosing object is the immediate parent if you are nesting the objects.

```
var f = {
    name: "f",
    func: function () {
        return this;
    }
};
console.log(f.func());
//prints -
//[object Object] {
//  func: function () {
//        return this;
//    },
//  name: "f"
//}
```

- When there is no context – a function when invoked without any object does not get any context. By default it is bound to the global context. When you use 'this' inside such a function, it is also bound to the global context.
- When 'this' is used inside a constructor function – as we saw earlier, when a function is called with a 'new' keyword, it acts as a constructor. In case of a constructor, 'this' points to the object being constructed. In the example below, f() is used as a constructor (because it's invoked with a 'new' keyword) and hence 'this' is pointing to the new object being created. So when we say this.member = "f", the new member is added to the object being created, in this case that object happens to be 'o'.

```
var member = "global";
function f()
{
    this.member = "f";
}
var o= new f();
console.log(o.member); // f
```

Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: “
Deleted: ”
Deleted: ‘
Deleted: ’
Deleted: "
Deleted: "
Deleted: "
Deleted: "

We saw that instance properties take precedence when same property is defined both as an instance property and a prototype property. It is easy to visualize that when a new object is created, the properties of the constructor's prototype are copied over. However, that is not a correct assumption. What actually happens is that the prototype is attached to the object and referred when any property of that object is referred. Essentially, when a property is referenced on an object:

- The object is checked for the existence of the property. If it's found, the property is returned, else,
- The associated prototype is checked, if the property is found, it is returned, else an 'undefined' is returned

This is an important understanding because, in JavaScript, the following code actually works perfectly:

```
function Player() {
  isAvailable=false;
}
var crazyBob = new Player();
Player.prototype.isAvailable = function() {
  return isAvailable;
};
console.log(crazyBob.isAvailable()); //false
```

This code is a slight variation of the earlier example. We are creating the object first and then attaching the function to it's prototype. When you eventually call the method isAvailable() on the object, it is available since if it's not found in the object, JavaScript goes to its prototype to search for it. Think of this as a "hot code loading" – when used properly, this ability can give you incredible power to extend the basic object framework even after the object is created.

If you are familiar with object-oriented programming already, you must be wondering if we can control the visibility and access of the members of an object. As we discussed earlier, JavaScript does not have classes. In programming languages like Java, you have access modifiers like 'private', 'public' that let you control visibility of the class members, in JavaScript, we can achieve something similar by using function scope.

- You can declare private variables using 'var' keyword within a function. They can be accessed by private functions or privileged methods

Deleted: ’

Deleted: ’

Deleted: ‘

Deleted: ’

Deleted: ’

Deleted: because of the

Deleted: its

Comment [MP7]: Do let me know if this sentence has the same meaning that you intend to convey.

Deleted: “

Deleted: ”

Deleted:

Deleted: ‘

Deleted: ’

Deleted: ‘

Deleted: ’

Deleted: ‘

Deleted: ’

- Private functions may be declared inside object's constructor and can be called by privileged methods
- Privileged methods can be declared by `this.method=function() {}`
- Public methods are declared with Class.prototype.method=function(){}
- Public properties can be declared by this.property and can be accessed from outside the object

The following example shows several ways of doing this:

```
function Player(name,sport,age,country){

    this.constructor.noOfPlayers++;

      // Private Properties and Functions
      // Can only be viewed, edited or invoked by privileged members
    var retirementAge = 40;
    var available=true;
    var playerAge = age?age:18;
      function isAvailable(){ return available &&
(playerAge<retirementAge); }
    var playerName=name ? name :"Unknown";
    var playerSport = sport ? sport : "Unknown";

    // Privileged Methods
      // Can be invoked from outside and can access private members
      // Can be replaced with public counterparts
      this.book=function(){
            if (!isAvailable()){
                  this.available=false;
        } else {
          console.log("Player is unavailable");
        }
      };
    this.getSport=function(){ return playerSport; };
```

Deleted: '

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Deleted: "

```
    // Public properties, modifiable from anywhere
      this.batPreference="Lefty";
      this.hasCelebGirlfriend=false;
    this.endorses="Super Brand";
}

// Public methods - can be read or written by anyone
// Can only access public and prototype properties
Player.prototype.switchHands = function(){
this.batPreference="righty"; };
Player.prototype.dateCeleb = function(){
this.hasCelebGirlfriend=true; } ;
Player.prototype.fixEyes = function(){ this.wearGlasses=false; };

// Prototype Properties - can be read or written by anyone (or
overridden)
Player.prototype.wearsGlasses=true;

// Static Properties - anyone can read or write
Player.noOfPlayers = 0;



(function PlayerTest(){
    //New instance of the Player object created.
      var cricketer=new Player("Vivian","Cricket",23,"England");
      var golfer =new Player("Pete","Golf",32,"USA");
    console.log("So far there are " + Player.noOfPlayers + " in the
guild");

    //Both these functions share the common
'Player.prototype.wearsGlasses' variable
      cricketer.fixEyes();
    golfer.fixEyes();
```

Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: we
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: '
Deleted: '

```
    cricketer.endorses="Other Brand";//public variable can be
updated


    //Both Player's public method is now changed via their prototype
    Player.prototype.fixEyes=function(){
            this.wearGlasses=true;
    };
    //Only Cricketer's function is changed
    cricketer.switchHands=function(){
            this.batPreference="undecided";
    };

})();
```

Let's understand a few important concepts from this example.

- Variable `retirementAge` is a private variable that has no privileged method to get or set it's value
- Variables `country` is a private variable created as a constructor argument. Constructor arguments are available as private variables to the object.
- When we called `cricketer.switchHands()`, it was only applied to the cricketer and not to both the players although it's a prototype function of the Player itself
- Private functions and privileged methods are instantiated with each new object created – in our example, new copies of `isAvailable`() and `book`() would be created for each new player instance we create. On the other hand, only one copy of public methods are created and shared between all instances. This can mean a bit of performance gain. If you don't 'really' need to make something private, think about keeping it public.

Inheritance

Inheritance is an important concept of Object Oriented programming. It is common to have a bunch of objects implementing same methods. It is also common to have almost similar object definition with differences in a few methods. Inheritance is very useful in

Deleted: `//Sets a public variable, but does not overwrite private ''sport'' variable.` ... [2]

Deleted: "

Deleted: "

Deleted: '

Deleted: '

Deleted: "

Deleted: "

Deleted: '

Deleted: '

Deleted: '

Deleted: <#>When we called `golfer.sport = ""Soccer""`, it did not modify the private variable `sport` which was assigned to value `golf` when the object was created. So you can essentially have both public and private variables with the same name, it would be utterly stupid however to do so and you should refrain from such adventures .

Deleted: '

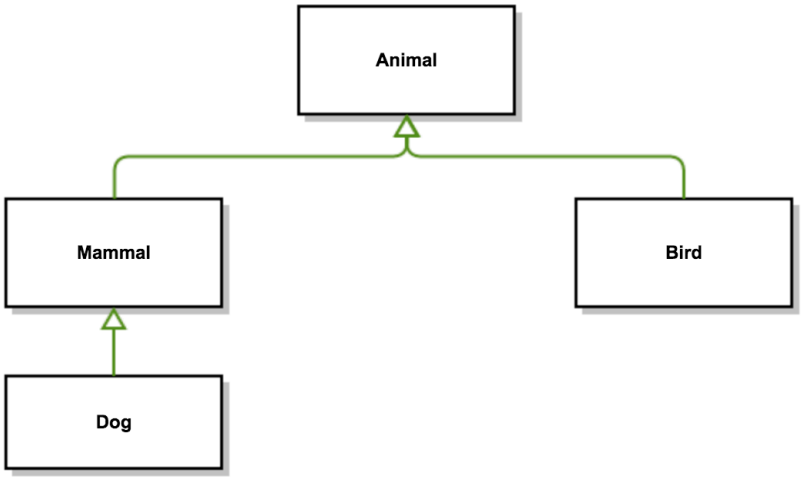
Deleted: ‘

Deleted: '

Formatted: Heading 1;Heading 1 [PACKT]

Deleted: in of

promoting code reuse. We can look at the following classic example of inheritance relation.



Here, you see that from the generic Animal class, we derive more specific classes like Mammal and Bird based on specific characteristics. Both mammal and bird classes do have the same template of an 'Animal' however they also define behaviors and attributes specific to them. Eventually we derive a very specific mammal, a dog. A dog has common attributes and behaviors from an animal class and a mammal class, while it adds specific attributes and behaviors of a dog. This can go on to add complex inheritance relationship. Traditionally, inheritance is used to establish or describe an "IS-A" relationship. For example, a dog "IS-A" mammal. This is what we know as 'classical inheritance'. You would have seen such relationships in Object Oriented languaes like C++ and Java. JavaScript has a completely different mechanism to handle inheritance. JavaScript is class-less language and uses prototypes for inheritance. Prototypal inheritance is very different in nature needs thorough understanding. Classical and prototypal inheritance are very different in nature and needs careful study.

In classical inheritance, instances inherit from a class blueprint, and create sub-class relationships. You can't invoke instance methods on a class definition itself. You need to

Deleted: Mammel
Deleted: ‘
Deleted: ’
Deleted: “
Deleted: ”
Deleted: “
Deleted: ”
Deleted: ‘
Deleted: ’
Deleted: and
Deleted: ’

create an instance and then invoke methods on that instance. In prototypal inheritance on the other hand, instances inherit from other instances.

As far as inheritance is concerned, JavaScript only uses objects. As we discussed earlier, each object has link to another object called its prototype. That prototype object in turn has a prototype of its own, and so on until an object is reached with null as its prototype. `null`, by definition, has no prototype, and acts as the final link in this prototype chain.

To understand prototype chains better, let's consider the following example.

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
function Child() {}
Child.prototype = {cry: Person.prototype.cry};
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //false
console.log(aChild instanceof Object); //true
```

Here, we define a Person and then a Child – a child IS-A person. We also copy 'cry' property of a Person to the Child's 'cry' property. When we try to see this relationship using `instanceof`, we soon realize that just by copying a behavior, we could not really make Child an instance of a Person. `aChild instanceof Person` fails. This is just copying or masquerading, not inheritance. Even if we copy all the properties of a `Person` to the `Child`, we won't be inheriting from `Person`. This is usually a bad idea and is shown here only for illustrative purposes. We want to derive a prototype chain – an IS-A relationship, a real inheritance where we can say that a Child IS-A person. We want to create a chain like – aChild IS-A Person IS-A Mammal IS-A Animal IS-A Object. In JavaScript, this is done by using an instance of an object as prototype. Something like:

```
SubClass.prototype = new SuperClass();
Child.prototype = new Person();
```

Let's modify the earlier example.

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
```

Formatted: English (US)

Deleted: '

Deleted: "

Deleted: "

Deleted: '

Deleted: '

Deleted: '

Deleted: '

Deleted: '

Deleted: masquarading

Deleted: –

Formatted: English (US)

Formatted: English (US)

Deleted: '

Formatted: English (US)

Deleted: '

Deleted: "

Deleted: "

```
function Child() {}
Child.prototype = new Person();
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //true
console.log(aChild instanceof Object); //true
```

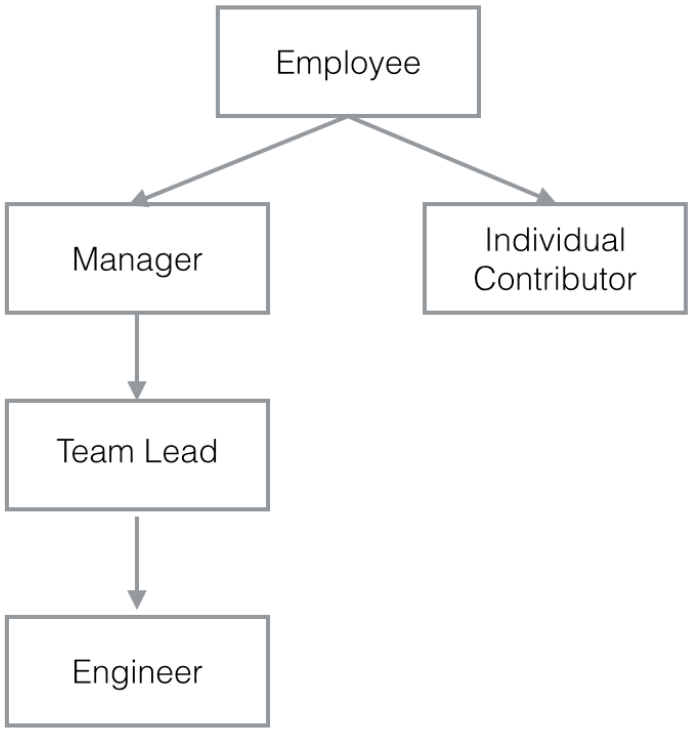
The changed line uses an instance of a Person as the prototype of a Child. This is an important distinction from the earlier method. Here we are declaring that a Child IS-A Person.

We discussed about how JavaScript looks for a property up the prototype chain till it reaches Object.prototype. Let us discuss the concept of prototype chains in detail and try to design the following employee hierarchy.

Comment [MP10]: What concept are we referring here to?

Deleted: more

Deleted: . Let us



This is a typical pattern of inheritance. A manager IS-A(n) employee. Manager has common properties inherited from an Employee. It can have an array of reportees. An individual contributor is also based on an employee but he does not have any reportees. A team-lead is derived from a manager with a few functions different from a manager. What essentially we are doing is that each child is deriving properties from its parent (Manager being the parent and Team Lead being the child)

Let us see how we can create this hierarchy in JavaScript. Let's define our Employee type.

```
function Employee() {
  this.name = '';
```

Deleted: '

Deleted: '

Deleted: '

```
  this.dept = 'None';
  this.salary = 0.00;
}
```

There is nothing special about these definitions. Employee object contains three properties – name, salary and dept. Next we define a Manager. This definition shows how to specify the next object in the inheritance chain.

```
function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);
```

In JavaScript, you can add a prototypical instance as the value of the prototype property of the constructor function. You can do so at any time after you define the constructor. In this example, there are two ideas we have not explored earlier. First, we are calling Employee.call(this) – If you are coming from Java background, this is analogous to super() method call inside constructor. The call() method calls a function with a specific object as its context (in this case it is the given 'this' value), in other words, call allows to specify which object will be referenced by the this keyword when the function will be executed. Like super() in Java, calling parentObject.call(this) is necessary to correctly initialize the object being created.

The other thing we see is Object.create() instead of calling a 'new'. Object.create() creates an object with a specified prototype. When we do a 'new Parent()' – the constructor logic of the parent is called. In most cases what we want to do is to like Child.prototype to be an object which is linked via its prototype to Parent.prototype. If the parent constructor contains additional logic specific to the parent, we don't want to run that while creating the child object. This can cause very difficult to find bugs. Object.create() creates the same prototypal link between child and parent as 'new' operator **without calling the parent constructor.**

To have a side-effect free and accurate inheritance mechanism, we have to make sure that:

Deleted: '
Deleted: '
Deleted: two
Comment [MP11]: Is salary and department a single property. If not just rectify the number of properties to 3
Deleted: s
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: correct
Deleted: to
Deleted:
Deleted: both of them to

- Setting the prototype to an instance of the parent initializes the prototype chain (inheritance), this is done only once (since the prototype object is shared).
- Calling the parent's constructor initializes the object itself, this is done with every instantiation (you can pass different parameters each time you construct it).

With this understanding in place, let's define the rest of the Objects.

```
function IndividualContributor() {
  Employee.call(this);
  this.active_projects = [];
}
IndividualContributor.prototype =
Object.create(Employee.prototype);


function TeamLead() {
   Manager.call(this);
   this.dept = "Software";
   this.salary = 100000;
}
TeamLead.prototype = Object.create(Manager.prototype);


function Engineer() {
   TeamLead.call(this);
   this.dept = "JavaScript";
   this.desktop_id = "8822" ;
   this.salary = 80000;
}
Engineer.prototype = Object.create(TeamLead.prototype);
```

Based on this hierarchy, we can instantiate these objects.

```
var genericEmployee = new Employee();
console.log(genericEmployee);
```

You should see the following output for the code snippet above:

```
[object Object] {
  dept: "None",
```

Formatted: Outline numbered + Level: 1 + Numbering Style: Bullet + Aligned at: 0,63 cm + Tab after: 1,27 cm + Indent at: 1,27 cm

Deleted: '

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Deleted: "

Deleted: "

```
  name: "",
  salary: 0
}
```

A genericEmployee has dept assigned to 'None' (as specified in the default value) and the rest of the properties are also assigned as the default ones.

Next, we instantiate a manager, we can provide specific values as follows-

```
var karen = new Manager();
karen.name = "Karen";
karen.reports = [1,2,3];
console.log(karen);
```

You will see the following output:

```
[object Object] {
  dept: "None",
  name: "Karen",
  reports: [1, 2, 3],
  salary: 0
}
```

For a TeamLead, the 'reports' property is derived from the base class (Manager) in this case.

```
var jason = new TeamLead();
jason.name = "Json";
console.log(jason);
```

You will see the following output:

```
[object Object] {
  dept: "Software",
  name: "Json",
  reports: [],
  salary: 100000
}
```

When JavaScript processes the new operator, it creates a new object and passes this object as the value of the 'this' to the parent - TeamLead constructor. The constructor function sets the value of the 'projects' property, and implicitly sets the value of the internal `__proto__` property to the value of `TeamLead.prototype`. The `__proto__` property determines the prototype chain used to return property values. This process does not set values for properties inherited from the prototype chain in jason object. When the value of a property is read, JavaScript first checks to see if the value exists in that object.

Deleted: "
Deleted: "
Deleted: ‘
Deleted: ’
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: ‘
Deleted: ’
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: encounters
Deleted: ‘
Deleted: ’
Deleted: ‘
Deleted: ’
Comment [VA13]: This is very standard sentence. Not sure if this needs to be modfied.
Comment [MM14R13]: To the reviewers: Please check if this line can be modified.

If the value does exist, that value is returned. If the value is not there, JavaScript checks the prototype chain using the `__proto__` property. Having said that, what happens when you do this:

```
Employee.prototype.name = "Undefined";
```

It does NOT propagate to all instances of `Employee`. This is because when you create an instance of the `Employee` object, that instance gets a local value for the name. When you set the `TeamLead` prototype by creating a new `Employee` object, `TeamLead.prototype` has a local value for the name property. Therefore, when JavaScript looks up the name property of the `jason` object, which is an instance of `TeamLead`), it finds the local value for that property in `TeamLead.prototype`. It does not try to do further lookups up the chain to `Employee.prototype`.

If you want the value of a property changed at runtime and have the new value be inherited by all descendants of the object, you cannot define the property in the object's constructor function. To achieve that, you need to add it to the constructor's prototype. For example, let's revisit the earlier example but with a slight change:

```
function Employee() {
  this.dept = 'None';
  this.salary = 0.00;
}
Employee.prototype.name = '';
function Manager() {
  this.reports = [];
}
Manager.prototype = new Employee();
var sandy = new Manager();
var karen = new Manager();

Employee.prototype.name = "Junk";

console.log(sandy.name);
console.log(karen.name);
```

Deleted: it
Deleted: locally
Deleted: "
Deleted: "
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: "
Deleted: "

You will see that 'name' property of both sandy and karen has changed to "Junk". This is because 'name' property is declared outside the constructor function. So, when you change the value of 'name' on Employee's prototype, it propages to all descendents. In this example, we are modifying Employee's prototype **after** sandy and karen objects are created. If you changed the prototype **before** sandy and karen objects were created, the value would have still changed to 'Junk'.

All native JavaScript objects - Object, Array, String, Number, RegExp, and Function- have prototype properties that can be extended. This means that we can extend the functionality of the language itself. For example, the following snippet, extends String object to add a method reverse() to reverse a string. This method does not exist in the native String object but by manipulating String's prototype, we add this method to String.

```
String.prototype.reverse = function() {
    return Array.prototype.reverse.apply(this.split('')).join('');
};
var str = 'JavaScript';
console.log(str.reverse()); //"tpircSavaJ"
```

Though this is a very powerful technique, care should be taken not to overuse it. Please refer to - http://perfectionkills.com/extending-native-builtins/ - to understand pitfalls of extending native builtins and what care should be taken if you intend to do so.

Getters and Setters

Getters are convenient methods to get value of specific properties and as the name suggests, setters are methods that sets value of property. Often you may want to derive a value based on some other values. Traditionally, getters and setters used to be functions like these:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  setLastName: function(_lastname){
    this.lastname= _lastname;
  },
  setFirstName: function (_firstname){
```

Deleted: ‘
Deleted: ’
Deleted: “
Deleted: ”
Deleted: ‘
Deleted: ’
Deleted: When
Deleted: ‘
Deleted: ’
Deleted: ’
Deleted: ’
Deleted: ‘
Deleted: ’
Deleted: ’
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: "
Deleted: "
Formatted: Heading 1;Heading 1 [PACKT]
Deleted:
Deleted: "
Deleted: "
Deleted: "
Deleted: "

```
    this.firstname= _firstname;
  },
  getFullName: function (){
    return this.firstname + ' '+ this.lastname;
  }
};
person.setLastName('Newton');
person.setFirstName('Issac');
console.log(person.getFullName());
```

As you can see, setLastName(), setFirstName() and getFullName() are functions used to do get and set of properties. Fullname is a derived property by concatenating firstname and lastname properties. This is a very common usecase and ECMAScript5 now provides default syntax for getters and setters.

The following example shows how getters and setters are created using object literal syntax in ECMAScript5.

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  get fullname() {
    return this.firstname +" "+this.lastname;
  },
  set fullname(_name){
    var words = _name.toString().split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
};
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

Another way of declaring getters and setters are by using Object.defineProperty() method.

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
};
Object.defineProperty(person, 'fullname', {
    get: function() {
        return this.firstname + ' ' + this.lastname;
    },
    set: function(name) {
        var words = name.split(' ');
        this.firstname = words[0];
        this.lastname = words[1];
    }
});
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

In this method, you can call Object.defineProperty() even after the object is created.

Now that you have tasted object-oriented flavor of JavaScript, we would go through a bunch of very useful utility methods provided by Underscore.js. These methods will make common operations on objects very easy. We discussed installation and basic usage of Underscore.js in previous chapter.

- keys(): This method retrieves names of object's own enumerable properties. Please note that this function does not traverse up the prototype chain.

```
var _ = require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
```

```
};
console.log(_.keys(testobj));
//[ 'name', 'age', 'profession' ]
```

- allKeys(): This method retrieves names of object's own and inherited properties.

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
}
Scientist.prototype.married = true;
aScientist = new Scientist();
console.log(_.keys(aScientist)); //[ 'name' ]
console.log(_.allKeys(aScientist));//[ 'name', 'married' ]
```

- values(): This method retrieves values of object's own properties.

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
}
Scientist.prototype.married = true;
aScientist = new Scientist();
console.log(_.values(aScientist)); //[ 'Albert' ]
```

- mapObject(): This method transforms the value of each property in the object.

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
  this.age = 90;
}
aScientist = new Scientist();
var lst = _.mapObject(aScientist, function(val,key){
  if(key==="age"){
    return val + 10;
  } else {
    return val;
  }
```

Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: `
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: `
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: "
Deleted: "

```
});
console.log(lst); //{ name: 'Albert', age: 100 }
```

- functions() : Returns a sorted list of the names of every method in an object - the name of every function property of the object.
- pick() : This function returns copy of the object, filtered to only the values of the keys provided .

```
var _ = require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
};
console.log(_.pick(testobj, 'name','age')); //{ name:
'Albert', age: 90 }
console.log(_.pick(testobj, function(val,key,object){
  return _.isNumber(val);
})); //{ age: 90 }
```

- omit() : This function is an invert of pick() - it returns a copy of the object, filtered to omit the values for the specified keys.

Summary

By allowing for the greater degree of control and structure that object-orientation can bring to the code, JavaScript applications can improve in clarity and quality. JavaScript object-orientation is based on function prototypes and prototypal inheritance. These two ideas can provide an incredible amount of wealth to developers.

In this chapter, we saw basic object creation and manipulation. We looked at how constructor functions are used to create objects. We dived into prototype chains and how inheritance operates on the idea prototype chains. These foundations will be used to build your knowledge of JavaScript patterns we are going to explore in the next chapter.

Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Deleted: '
Comment [IM15]: This part does not pertain to JavaScript itself, so it can be removed from the book.
Formatted: Heading 1;Heading 1 [PACKT]
Comment [MP16]: Please mention what we would be covering in the next chapter as well

Page 6: [1] Formatted **Manasi Pandire** **11/08/15 17:15**

Font:Lucida Console, 9,5 pt, Font color: R,G,B (116,121,89), English (US)

Page 11: [2] Deleted **Ivano Malavolta** **21/10/15 22:05**

```
    //Sets a public variable, but does not overwrite private ''sport'' variable.
      golfer.sport = ""Soccer"";
      console.log(golfer.getSport()); //Returns ''Golf'' from private ''sport'' and
not the public variable set earlier
```