



# Understanding @Output and EventEmitter in Angular

by Dhananjay Kumar MVB · Jan. 05, 18 · Web Dev Zone · Tutorial

Data is at the heart of a lot of development projects. And data aggregation, manipulation and writing can be really time consuming. Then the data changes and you have to do it all again when Qlik Core, an analytics component by Qlik, can help. [Try it today.](#)

Presented by Qlik

In Angular, a component can emit an event using **@Output** and **EventEmitter**. Both are parts of the **@angular/core**.

Confused by the jargon? Let's simplify it together. Consider the **AppChildComponent** as shown below:

## appchild.component.ts

```
1  import { Component, Input, EventEmitter, Output } from '@angular/core';
2
3  @Component({
4      selector: 'app-child',
5      template: `<button class='btn btn-primary' (click)="handleclick()">Click me</butt
6  })
7  export class AppChildComponent {
8
9      handleclick() {
10
11          console.log('hey I am clicked in child');
12      }
13  }
```

There's a button in the AppChildComponent template which is calling the function handleclick. Let's use the app-child component inside the App Component as shown below:

## appcomponent.ts

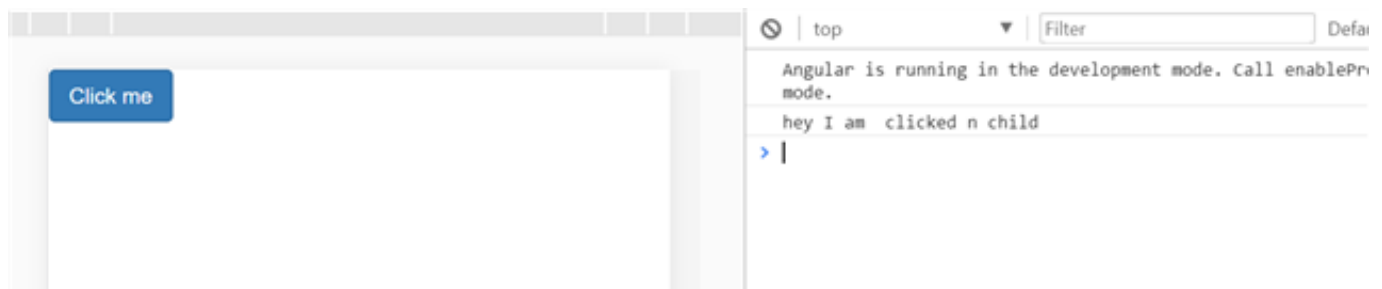
```
1  import { Component, OnInit } from '@angular/core';
2  @Component({
```

```

3     selector: 'app-root',
4     template: `<app-child></app-child>`
5   })
6   export class AppComponent implements OnInit {
7
8     ngOnInit() {
9
10    }
11  }

```

Here we're using **AppChildComponent** inside **AppComponent**, thereby creating a parent/child kind of relationship, in which **AppComponent** is the parent and **AppChildComponent** is the child. When we run the application with a button click, you'll see this message in the browser console:



So far, it's very simple to use event binding to get the button to call the function in the component. Now, let's tweak the requirement a bit. What if you want to execute a function of **AppComponent** on the click event of a button inside **AppChildComponent**?

To do this, you will have to emit the button click event from **AppChildComponent**. Import **EventEmitter** and **Output** from **@angular/core**.

Here we are going to emit an event and pass a parameter to the event. Consider the code below:

### appchild.component.ts

```

1  import { Component, EventEmitter, Output } from '@angular/core';
2
3  @Component({
4    selector: 'app-child',
5    template: `<button class='btn btn-primary' (click)="valueChanged()">Click me</but
6  })
7  export class AppChildComponent {
8
9    @Output() valueChange = new EventEmitter();
10    Counter = 0;
11
12    valueChanged() { // You can give any function name
13
14      this.counter = this.counter + 1;

```

```

15         this.valueChange.emit(this.counter);
16     }
17 }

```

Right now, we are performing the following tasks in the **AppChildComponent** class:

1. Creating a variable called `counter`, which will be passed as the parameter of the emitted event.
2. Creating an EventEmitter, `valueChange`, which will be emitted to the parent component on the click event of the button.
3. Creating a function named `valueChanged()`. This function is called on the click event of the button, and inside the function event `valueChange` is emitted.
4. While emitting the `valueChange` event, the value of the counter is passed as a parameter.

In the parent component, **AppComponent**, the child component, **AppChildComponent**, can be used as shown in the code below:

### appcomponent.ts

```

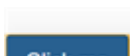
1  import { Component, OnInit } from '@angular/core';
2  @Component({
3      selector: 'app-root',
4      template: `<app-child (valueChange)='displayCounter($event)'></app-child>`
5  })
6  export class AppComponent implements OnInit {
7      ngOnInit() {
8
9      }
10     displayCounter(count) {
11         console.log(count);
12     }
13 }

```

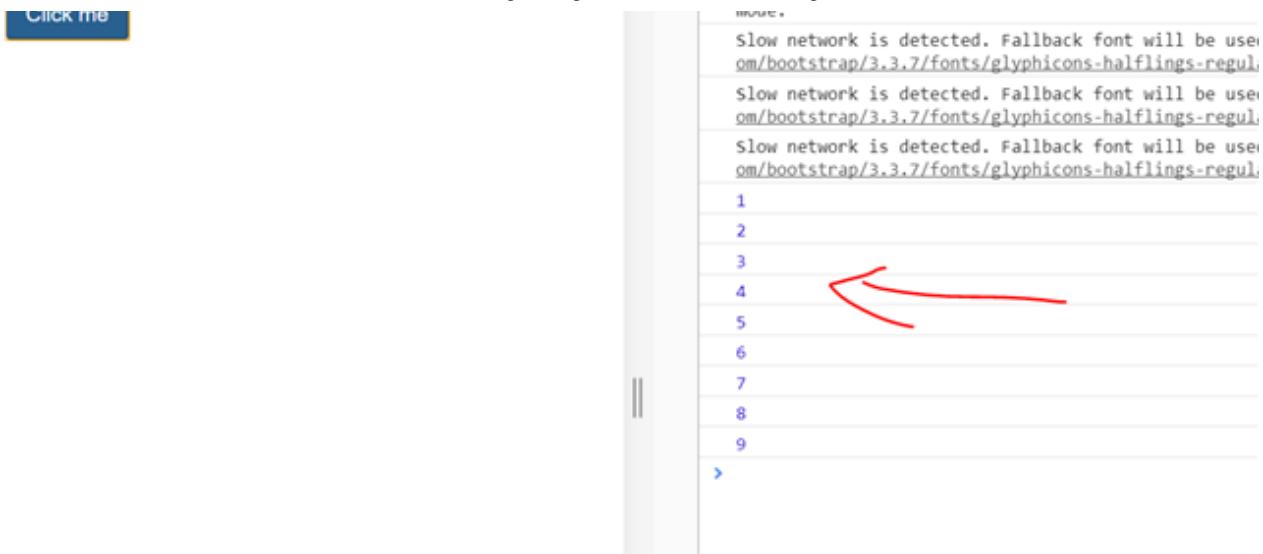
Right now, we are performing the following tasks in the **AppComponent** class:

1. Using `<app-child>` in the template.
2. In the `<app-child>` element, using event binding to use the `valueChange` event.
3. Calling the `displayCounter` function on the `valueChange` event.
4. In the `displayCounter` function, printing the value of the counter passed from the **AppChildComponent**.

As you can see, the function **AppComponent** is called on the click event of the button placed on the **AppChildComponent**. This can be done with `@Output` and `EventEmitter`. When you run the application and click the button, you can see the value of the counter in the browser console. Each time you click on the button, the counter value is increased by 1.



Angular is running in the development mode. Call en-  
mode



## A Real-Time Example

Let's take a real-time example to find out how @Output and EventEmitter can be more useful. Consider that **AppComponent** is rendering a list of products in a tabular form as shown in the image below:

Products			
Id	Title	Price	Stock
1	Screw Driver	400	11
2	Nut Volt	200	5
3	Resistor	78	45
4	Tractor	20000	1
5	Roller	62	15

To create the product table above, we have a very simple **AppComponent** class with only one function: to return a list of products.

```

1  export class AppComponent implements OnInit {
2      products = [];
3      title = 'Products';
4      ngOnInit() {
5          this.products = this.getProducts();
6      }
7      getProducts() {
8          return [
9              { 'id': '1', 'title': 'Screw Driver', 'price': 400, 'stock': 11 },
10             { 'id': '2', 'title': 'Nut Volt', 'price': 200, 'stock': 5 },
11             { 'id': '3', 'title': 'Resistor', 'price': 78, 'stock': 45 },
12             { 'id': '4', 'title': 'Tractor', 'price': 20000, 'stock': 1 },
13             { 'id': '5', 'title': 'Roller', 'price': 62, 'stock': 15 },
14         ];
15     }

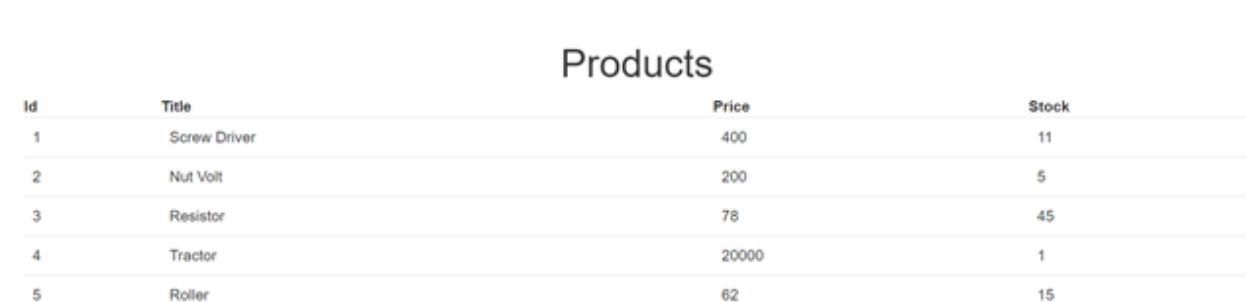
```

```
16 }
```

In the `ngOnInit` lifecycle hook, we are calling the `getPrdoducts()` function and assigning the returned data to the `products` variable so it can be used in the template. There, we are using the `*ngFor` directive to iterate through the array and display the products. See the code below:

```
1 <div class="container">
2   <br />
3   <h1 class="text-center">{{title}}</h1>
4   <table class="table">
5     <thead>
6       <th>Id</th>
7       <th>Title</th>
8       <th>Price</th>
9       <th>Stock</th>
10    </thead>
11    <tbody>
12      <tr *ngFor="let p of products">
13        <td>{{p.id}}</td>
14        <td>{{p.title}}</td>
15        <td>{{p.price}}</td>
16        <td>{{p.stock}}</td>
17      </tr>
18    </tbody>
19  </table>
20 </div>
```

With this code, products are rendered in a table as shown in the image below:



Id	Title	Price	Stock
1	Screw Driver	400	11
2	Nut Volt	200	5
3	Resistor	78	45
4	Tractor	20000	1
5	Roller	62	15

Now let's say we want to add a new column with a button and input box as shown in the image below:



Id	Title	Price	Stock	
1	Screw Driver	400	7	<input type="text"/> <button>Change Stock Value</button>
2	Nut Volt	200	5	<input type="text"/> <button>Change Stock Value</button>
3	Resistor	78	45	<input type="text"/> <button>Change Stock Value</button>
4	Tractor	20000	1	<input type="text"/> <button>Change Stock Value</button>



Our requirements are as follows:

1. If the value of **stock** is more than 10 then the button color should be green.
2. If the value of **stock** is less than 10 then the button color should be red.
3. The user can enter a number in the input box, which will be added to that particular stock value.
4. The color of the button should be updated on the basis of the changed value of the product stock.

To achieve this, let us create a new child component called **StockStatusComponent**. Essentially, in the template of **StockStatusComponent**, there is one button and one numeric input box. In **StockStatusComponent**:

1. We need to read the value of **stock** passed from **AppComponent**. For this, we need to use **@Input**
2. We need to emit an event so that a function in **AppComponent** can be called on the click of the **StockStatusComponent**. For this, we need to use **@Output** and **EventEmitter**.

Consider the code below:

### stockstatus.component.ts

```
import { Component, Input, EventEmitter, Output, OnChanges } from '@angular/core';

1
2 @Component({
3   selector: 'app-stock-status',
4   template: `<input type='number' [(ngModel)]='updatedstockvalue' /> <button class='
5     [style.background]='color'
6     (click)="stockValueChanged()">Change Stock Value</button> `
7 })
8 export class StockStatusComponent implements OnChanges {
9
10   @Input() stock: number;
11   @Input() productId: number;
12   @Output() stockValueChange = new EventEmitter();
13   color = '';
14   updatedstockvalue: number;
15   stockValueChanged() {
16     this.stockValueChange.emit({ id: this.productId, updatedstockvalue: this.updatedstockvalue });
17     this.updatedstockvalue = null;
18   }
19   ngOnChanges() {
20
21     if (this.stock > 10) {
```

```
22         this.color = 'green';
23     } else {
24         this.color = 'red';
25     }
26 }
27 }
```

Let's explore the above class line by line.

1. In the first line, we are importing everything that's required: `@Input`, `@Output`, etc.
2. In the template, there is one numeric input box which is bound to the **updatedStockValue** property using `[(ngModel)]`. We need to pass this value with an event to the **AppComponent**.
3. In the template, there is one button. On the click event of the button, an event is emitted to the **AppComponent**.
4. We need to set the color of the button on the basis of the value of product stock. So, we must use property binding to set the background of the button. The value of the color property is updated in the class.
5. We are creating two `@Input()` decorated properties - **stock** and **productId** - because the value of these two properties will be passed from **AppComponent**.
6. We are creating an event called **stockValueChange**. This event will be emitted to **AppComponent** on the click of the button.
7. In the **stockValueChanged** function, we are emitting the **stockValueChange** event and also passing the product id to be updated and the value to be added to the product stock value.
8. We are updating the value of the color property in the `ngOnChanges()` lifecycle hook because each time the stock value gets updated in the **AppComponent**, the value of the color property should be updated.

Here we are using the `@Input` decorator to read data from **AppComponent** class, which happens to be the parent class in this case. So to pass data from the parent component class to the child component class, use the `@Input` decorator.

In addition, we are using `@Output` with **EventEmitter** to emit an event to **AppComponent**. So to emit an event from the child component class to the parent component class, use **EventEmitter** with the `@Output()` decorator.

Therefore, **StockStatusComponent** is using both `@Input` and `@Output` to read data from **AppComponent** and emit an event to **AppComponent**.

## Modify AppComponent to Use StockStatusComponent

Let us first modify the template. In the template, add a new table column. Inside the column, the `<app-stock-status>` component is used.

```
1 <div class="container">
2     <br />
3     <h1 class="text-center">{{title}}</h1>
4     <table class="table">
```

```

5      <thead>
6      <th>Id</th>
7      <th>Title</th>
8      <th>Price</th>
9      <th>Stock</th>
10     </thead>
11     <tbody>
12         <tr *ngFor="let p of products">
13             <td>{{p.id}}</td>
14             <td>{{p.title}}</td>
15             <td>{{p.price}}</td>
16             <td>{{p.stock}}</td>
17             <td><app-stock-status [productId]='p.id' [stock]='p.stock' (stockValueChange)=$event</td>
18         </tr>
19     </tbody>
20 </table>
21 </div>

```

We are passing the value to **productId** and **stock** using property binding (remember, these two properties are decorated with `@Input()` in **StockStatusComponent**) and using event binding to handle the **stockValueChange** event (remember, this event is decorated with `@Output()` in **StockStatusComponent**).

```

1  productToUpdate: any;
2  changeStockValue(p) {
3      this.productToUpdate = this.products.find(this.findProducts, [p.id]);
4      this.productToUpdate.stock = this.productToUpdate.stock + p.updatdstockvalue;
5  }
6  findProducts(p) {
7      return p.id === this[0];
8  }

```

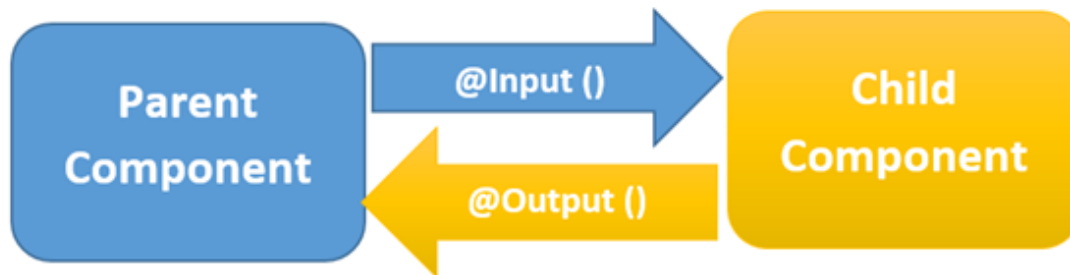
In the function, we are using the JavaScript **Array.prototype.find** method to find a product with a matched **productId** and then updating the stock count of the matched product. When you run the application, you'll get the following output:

Products					
Id	Title	Price	Stock		
1	Screw Driver	400	11	<input type="text"/>	Change Stock Value
2	Nut Volt	200	5	<input type="text"/>	Change Stock Value
3	Resistor	78	45	<input type="text"/>	Change Stock Value
4	Tractor	20000	1	<input type="text"/>	Change Stock Value
5	Roller	62	15	<input type="text"/>	Change Stock Value



When you enter a number in the numeric box and click on the button, you perform a task in the child component that updates the operation value in the parent component. Also, on the basis of the parent component value, the style is being changed in the child component. All this is possible using Angular @Input, @Output, and EventEmitter.

In summary:



Stay tuned for future articles where we go into more depth on other features of Angular!

---

From data concepts to cognitive services to machine learning, explore how to develop for , more Learning Paths at [Microsoft Ignite](#).

Presented by Microsoft

---

## Like This Article? Read More From DZone



**Real-World Angular Series, Part 2a: Authentication**



**Creating a Front-End for Your User Profile Store With Angular and TypeScript**



**Why You Should Use TypeScript for Developing Web Applications**



**Free DZone Refcard Getting Started With Java-Based CMS**

Topics: ANGULAR , WEB DEV , WEB APPLICATION DEVELOPMENT , TYPESCRIPT

Published at DZone with permission of Dhananjay Kumar , DZone MVB. [See the original article here.](#)



Opinions expressed by DZone contributors are their own.

# IEnumerable vs. IEnumerator

**by neeraj kumar · Aug 13, 19 · Web Dev Zone · Tutorial**

In this blog, I will try to explain the difference between IEnumerable and IEnumerator by using a simple example. First, we will understand the meaning of both of the terms and why we are using them.

What are IEnumerable and IEnumerator?

```
1 namespace System.Collections.Generic
2 {
3     //
4     // Summary:
5     //     Exposes the enumerator, which supports a simple iteration over a collectio
6     //     a specified type.
7     //
8     // Type parameters:
9     //     T:
10    //     The type of objects to enumerate.
11    public interface IEnumerable<out T> : IEnumerable
12    {
13        //
14        // Summary:
15        //     Returns an enumerator that iterates through the collection.
16        //
17        // Returns:
18        //     An enumerator that can be used to iterate through the collection.
19        IEnumerator<T> GetEnumerator();
20    }
21 }
```

The above code explains that IEnumerable internally implements the GetEnumerator() function of IEnumerator. From this, we can say IEnumerable works in the same way as IEnumerator, but there is a big difference between them that we will see with the help of example given below.

Both IEnumerable and IEnumerator are used to iterate through a collection object.

Let's look at this example:

```
1 static void Main(string[] args)
2 {
3     List<int> IDS = new List<int>();
4     IDS.Add(1);
5     IDS.Add(2);
6 }
```

```
5         IDS.Add(2);
6         IDS.Add(3);
7         IDS.Add(4);
8         IDS.Add(5);
9
10        IEnumerable<int> enumerableList = (IEnumerable<int>)IDS;
11
12        foreach (var id in enumerableList)
13        {
14            Console.WriteLine("id in enumerableList is {0}", id);
15        }
16        IEnumerator<int> enumeratorList = IDS.GetEnumerator();
17
18        while (enumeratorList.MoveNext())
19        {
20            Console.WriteLine("id in enumeratorList is {0}", enumeratorList.Curre
21        }
22
23        Console.ReadKey();
24    }
```

In the code block above, we created a List IDS of type integer and added some values to the list. Now, we create an object of IEnumerable<int> as enumerableList and initialize it with list IDS. We follow the same process for the IEnumerator. After creating the objects, we iterate through them. As we can see in the example, IEnumerator has some extra methods to perform operations, such as MoveNext() to find the next element and Current() to get the current value, as the same suggests.

Output for the cases will be the same; the only difference we can see here is we have written more code in IEnumerator for doing the same amount of work.

As we have seen IEnumerable internally implements IEnumerator, so using IEnumerable means we can write less code for the same work of iterating through a list.

Here's the question, "Why are we even using IEnumerator for iterating through any collection?"

Well, IEnumerator maintains the state for an iterating variable.

What does that mean, and what's its use? Modify the above example as:

```
1    class Program
2    {
3        static void Main(string[] args)
4        {
5            List<int> IDS = new List<int>();
6            IDS.Add(1);
```

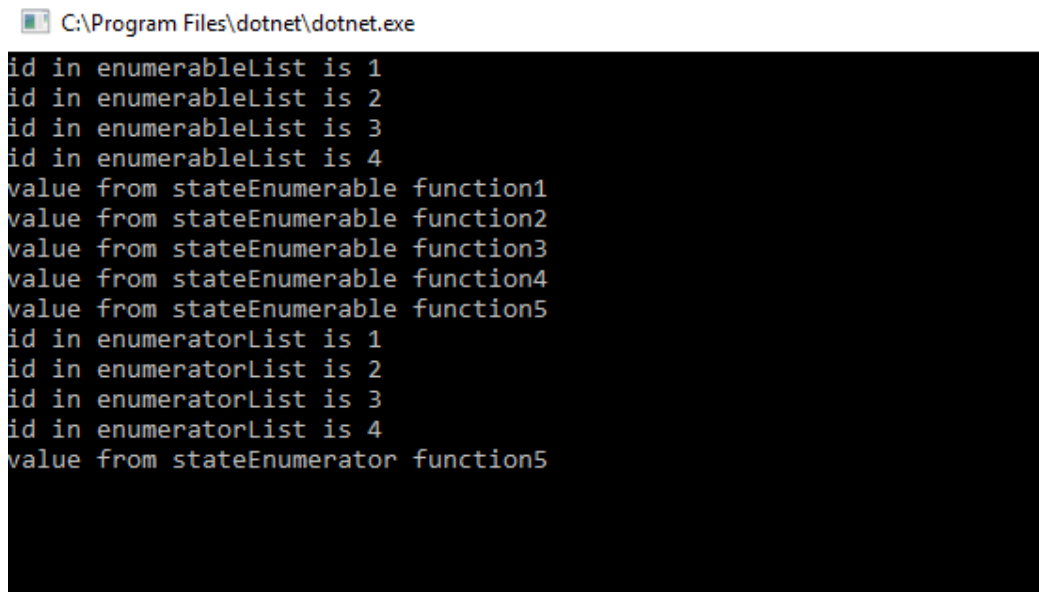
```
7         IDS.Add(2);
8         IDS.Add(3);
9         IDS.Add(4);
10        IDS.Add(5);
11
12        IEnumerable<int> enumerableList = (IEnumerable<int>)IDS;
13
14        foreach (var id in enumerableList)
15        {
16            Console.WriteLine("id in enumerableList is {0}", id);
17            if (id > 3)
18            {
19                StateEnumerable(enumerableList);
20                break;
21            }
22        }
23        IEnumerator<int> enumeratorList = IDS.GetEnumerator();
24
25        while (enumeratorList.MoveNext())
26        {
27            Console.WriteLine("id in enumeratorList is {0}", enumeratorList.Curre
28
29            if (enumeratorList.Current > 3)
30            {
31                StateEnumerator(enumeratorList);
32                break;
33            }
34        }
35
36
37        Console.ReadKey();
38    }
39    public static void StateEnumerable(IEnumerable<int> enumerableList)
40    {
41        foreach(int item in enumerableList)
42        {
43            Console.WriteLine("value from stateEnumerable function" + item);
44        }
45    }
46    public static void StateEnumerator(IEnumerator<int> enumerator)
47    {
48        while (enumerator.MoveNext())
49
```

```
50         {  
51             Console.WriteLine("value from stateEnumerator function" + enumerator.  
52         }  
53     }  
54 }
```

Now, we have added two new methods in our class `StateEnumerable(IEnumerable<int> enumerableList)` and `StateEnumerator(IEnumerable<int> enumerator)`. We call these methods in the loops of `enumerableList` and `enumeratorList` if the value of ID is greater than three.

Here, we are just interested in seeing how the iterating variable (looping variable as "var id in enumerable, id is iterating variable) state behaves in both the cases. Now, run this code and observe the outputs we get.

Output for the above code is :-



```
C:\Program Files\dotnet\dotnet.exe  
id in enumerableList is 1  
id in enumerableList is 2  
id in enumerableList is 3  
id in enumerableList is 4  
value from stateEnumerable function1  
value from stateEnumerable function2  
value from stateEnumerable function3  
value from stateEnumerable function4  
value from stateEnumerable function5  
id in enumeratorList is 1  
id in enumeratorList is 2  
id in enumeratorList is 3  
id in enumeratorList is 4  
value from stateEnumerator function5
```

So, here is the output, which explains the major difference between `IEnumerator` and `IEnumerable`. For a value of four for `i`, we called the method `StateEnumerable` function and `StsteEnumerator`

As `IEnumerator` is maintaining the state even after passing it, our other function started iterating from five, unlike `IEnumerable`, which started again from one.

## Conclusion:

So it depends on our condition. If we want to iterate through the list (or any collection) at one time, then `IEnumerable` is the best choice, but if we want to preserve the state of an iteration variable, then we should go with `IEnumerator`.

## Like This Article? Read More From DZone



**Tips for Migrating HSQL to PostgreSQL**



**Conversational AI: Design and Build a Contextual AI Assistant**



**Build a Xamarin Android Mobile Application Using Azure DevOps (CI)**



**Free DZone Refcard  
Getting Started With Java-Based CMS**

Topics: IENUMERABLE

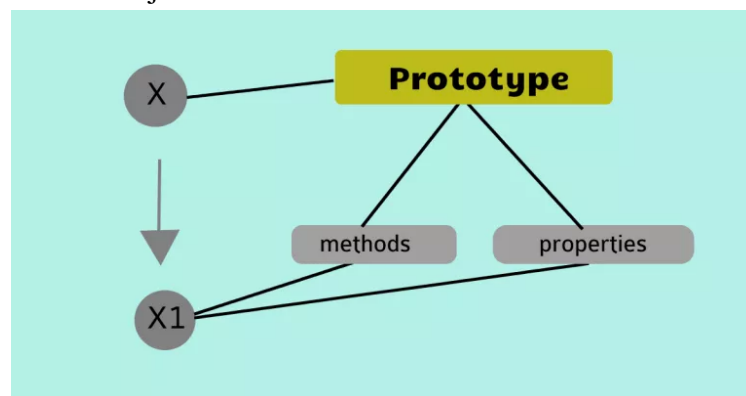
Opinions expressed by DZone contributors are their own.

# How Prototype Works in JavaScript?

by Atishay Khare · Aug 13, 19 · Web Dev Zone · Tutorial

In JavaScript, by default, every function has a property called prototype; this property by default is empty, and you can add properties and methods to it when you create an object from this function. The object inherits its properties and methods. For beginners, understanding the difference between prototype and `__proto__` has been difficult.

JavaScript does not have class implementation like Java or C#; in javascript, you create a constructor directly and then use this constructor to create an object.

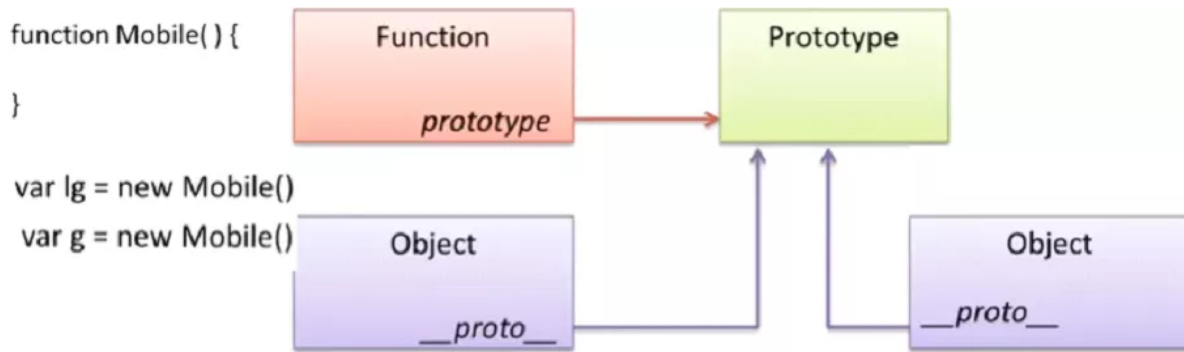


Here, X is a function and X1 is the object created from X.

Any function created in JavaScript is either a constructor or generic function. These two objects create a function object and prototype object; the function object holds a prototype property, and with the function name and `Function_Name.prototype`, we can access the prototype object properties.

When an object is created using the new keyword of the function/constructor, the JavaScript engine creates a new object of that function block which holds a property named `__proto__` which points to its function's prototype.

object of that function block, which holds a property named `__proto__`, which points to its function's prototype object.



If another object of that function is created using a new keyword again, another object is created, which holds a `__proto__` property, which again points to the function's prototype object. The same will be true for every object that is created for the function using the new keyword.

This can be checked by:

```

1  function Mobile() {
2
3  }
4
5  console.log(Mobile.prototype);
6
7  var lg = new Mobile();
8  console.log(lg.__proto__);
9
10 // verify
11 console.log(Mobile.prototype === lg.__proto__);
  
```

```

1  lg.__proto__ === Mobile.prototype // true
  
```

So, any property that is defined inside the constructor/function is accessible by the object of that function, which was created using the new keyword. So, when we try to access a property like `lg.a` it is first searched in the object of `lg`, and if it is not present, it is then searched in the prototype block of the constructor function. If it is not found even there, then it is undefined.

A property can also be defined inside the prototype block using:

```

1  functionName.prototype.propertyName = 'Value';
  
```

If a property with the same name is defined inside both the object as well as function prototype, then it is accessed from the object block. So, the first priority is of the object and then function prototype.

```
1  function Fun(){
2
3  this.a = 20;
4
5  }
6
7  Fun.prototype.a = 10;
8  let chk = new Fun();
9
10 console.log("The output is:- " + chk.a); //The output is 20
```

Also, the function object holds the function in it; whereas the object of that function also holds the function in its constructor. the same can be verified by:

```
1  console.log(Mobile === lg.__proto__.constructor); // true
```

The same holds true for the prototype of the function, as its constructor also holds the function in its constructor. This can also be verified by:

```
1  console.log(Mobile === Mobile.prototype.constructor); // true
```

so, basically, the function can be accessed using different ways.

---

## Like This Article? Read More From DZone



**Three Ways to Define Functions in JavaScript**



**Simplifying the Object.assign Method in JavaScript**



**Angular 2 – First Impressions [Compared to Angular 1]**



**Free DZone Refcard  
Getting Started With Java-Based CMS**



Published at DZone with permission of Atishay Khare . [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.