chapter 4

Constructors and Prototypes

ماهو constructor ؟

من أهم الأشياء التي عليك التفكير بها بعد إنشاء كلاس جديد , هي تسهيل طريقة خلق كائنات من هذا الكلاس .

من هنا جاءت فكرة constructor و الذي هو عبارة عن دالة لها نوع خاص , يتم استدعائها أثناء إنشاء كائن لتوليد قيم أولية للخصائص الموجودة فيه .

constructor هو ببساطة دالة تُستخدم مع new لإنشاء كائن .

ميزة constructor هي أن الكائنات التي تم إنشاؤها باستخدام نفس المُنشئ تحتوي على نفس properties and method.

بما أنه لا يمكن إنشاء كائن من كلاس إلا من خلال constructor , سيقوم مترجم جافا بتوليد constructor إفتراضي فارغ عنك إذا وجد أن الكلاس الذي قمت بتعريفه لا يحتوي على أي constructor .

* نقاط مهمة لابد من معرفتها عن constructor :

1_ أن constructor يأخذ نفس اسم class , قد يكون في class الواحد أكثر من constructor ولكن بمتغيرات مختلفة النوع أو العدد وهذا ما يسمى overloading .

2_ أن constructor لايرجع قيمة ولايمكن أن تكتب قبله كلمة void حتى .

3_ كل class داخل الجافا يوجد به constructor حتى لو لم يعرفه المبرمج , حيث يقوم الجافا بعمل constructor افتراضي.

4_ نستطيع استخدام أي access modifier عند كتابة constructor سواء كان

private , public , protect .

constructor يتم تحديده بنفس طريقة function , الاختلاف الوحيد هو أن أسماء constructor يجب أن تبدأ بحرف كبير ، لتمييزها عن function الأخرى .

على سبيل المثال :
(function Person) التالية فارغة .

```
function Person() {
  // intentionally empty
}
```

هذه function هي constructor لكن لا يوجد فرق في كتابة الجملة على الإطلاق بين هذه function وأي function أخرى .

الدليل على أن Person هو Constructor موجود في الاسم — نلاحظ أن الحرف الأول حرف كبير .

بعد ما يتم تعريف Constructor ، يمكنك البدء في إنشاء مثيلات ، مثل Person objects :

```
var person1 = new Person();
var person2 = new Person();
```

يمكنك حتى حذف الأقواس , عندما لا يكون لديك أي parameter لتمريرها :

```
var person1 = new Person;
var person2 = new Person;
```

يمكنك استخدام instanceof operator لاستنتاج نوع object :

```
console.log(person1 instanceof Person);     // true
console.log(person2 instanceof Person);     // true
```

في المثال السابق ==> نظرًا لأن person1 و person2 تم إنشاؤهما باستخدام constructor Person .
فإن instanceof operator يُرجع // true عندما يتحقق من أن هذه object هي مثيلات من Person .

يمكنك أيضًا التحقق من نوع object باستخدام constructor property :

```
console.log(person1.constructor ===  Person);     // true
console.log(person2.constructor ===  Person);     // true
```

يُنصح باستخدام instanceof للتحقق من نوع المثيل .
هذا لأن constructor property يمكنك الكتابة فوقه , لذلك قد لاتكون دقيقة تماماً .

الغرض من constructor هو تسهيل إنشاء المزيد من الكائنات بنفس الخصائص والأساليب , لذلك constructor function الفارغة ليست مفيدة جدًا .

مثال يوضح إضافة أي خصائص تريدها داخل constructor :

```
function Person(name) {
1-   this.name = name;
2-   this.sayName = function() {
         console.log(this.name);
     };
}
```

يقبل هذا المثال من (Constructor Person) متغير (parameter) واحد .
1- يخصصها ل Property name لهذا Object .
2- يضيف Constructor أيضًا (method) يحمل اسم sayName () إلى object .

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);      // "Nicholas"
console.log(person2.name);      // "Greg"
person1.sayName();              // outputs "Nicholas"
person2.sayName();              // outputs "Greg"
```

في المثال هذا استخدمنا constructor Person وأنشأنا منه مثيل له خاصية اسم (name property) .. كل object له خاصية الاسم الخاصة به ، لذلك يجب أن ترجع sayName () قيمًا مختلفة اعتمادًا على object الذي تستخدمه عليه.

note :

Return can also be called explicitly from within a constructor.

If the returned value is an object, the newly constructed object instance will be returned instead.

The newly formed object is utilized instead of the returned value if the returned value is a primitive.

The name property in this version of the Person constructor is an accessor property that stores the actual name in the name parameter.
Because named arguments behave as local variables, this is possible.
If you don't use new when calling constructors, you risk modifying the global object instead of the newly generated object.

على سبيل المثال ، يمكنك أيضًا استخدام

Object. selectProperty() داخل constructor
للمساعدة في تهيئة المثيل instance :

```
function Person(name) {
 Object.defineProperty(this, "name", {
       get: function() {
             return name;
      },
        set: function(newName) {
              name = newName;
      },
     enumerable: true,
     configurable: true
   });
this.sayName = function() {
console.log(this.name);
   };
 }
```

ضع في اعتبارك ما يحدث في الكود التالي :

```
var person1 = Person("Nicholas");          // note: missing "new"
console.log(person1 instanceof Person);    // false
console.log(typeof person1);               // "undefined"
console.log(name);                         // "Nicholas"
```

The value of this inside the constructor equals the global this object when Person is called as a function without new.

Because the Person constructor depends on new to provide a return value, the variable person1 has no value.

Person is just a function without a return statement without new.

The name provided to Person is saved in a global variable called name, which is created by the assignment to this.name.

The solution to this problem, as well as more complicated object composition patterns, is provided in Chapter 6.

ماهو Prototypes ؟

كلمة prototype تعني النموذج المبدئي للشيء .

فعندما تقوم بإنشاء كائن في javaScript تقوم اللغة بإنشاء prototype لهذا الكائن .

فأي object سوف تنشئه في javaScript سوف يكون له نموذج مبدئي ألا وهو الـ

prototype .

The hasOwnProperty() method , for example, is defined on the generic Object prototype, but it may be accessed as if it were an own property from any object ,
as seen in this example:

```
var book = {
  title: "The Principles of Object-Oriented JavaScript"
};
console.log("title" in book);    // true
console.log(book.hasOwnProperty("title"));   // true
console.log("hasOwnProperty" in book);   // true
console.log(book.hasOwnProperty("hasOwnProperty"));  // false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty"));  // true
```

في المثال السابق على الرغم من عدم وجود تعريف لـ hasOwnProperty () في book ،
لا يزال من الممكن الوصول إلى هذه method كـ book.hasOwnProperty () لأن
التعريف موجود في Object.prototype .

identifying a Prototype Property

يمكنك تحديد ما إذا كانت Property موجودة في Prototype باستخدام function مثل :

```
function hasPrototypeProperty(object, name) {
    return name in object && !object.hasOwnProperty(name);
}
console.log(hasPrototypeProperty(book, "title"));          // false
console.log(hasPrototypeProperty(book, "hasOwnProperty")); // true
```

If hasOwnProperty() returns false yet the property is in an object , the property is on the prototype.

The [[Prototype]] Property.....

[[Prototype]] is an internal property that an instance uses to keep track of its prototype.

This property is a reference to the prototype object used by the instance.

When you use new to create a new object, the constructor's prototype property is set to the new object's [[Prototype]] property.

Figure 4-1 shows how the [[Prototype]] property allows multiple instances of the same object type to refer to the same prototype, which can save time and code.

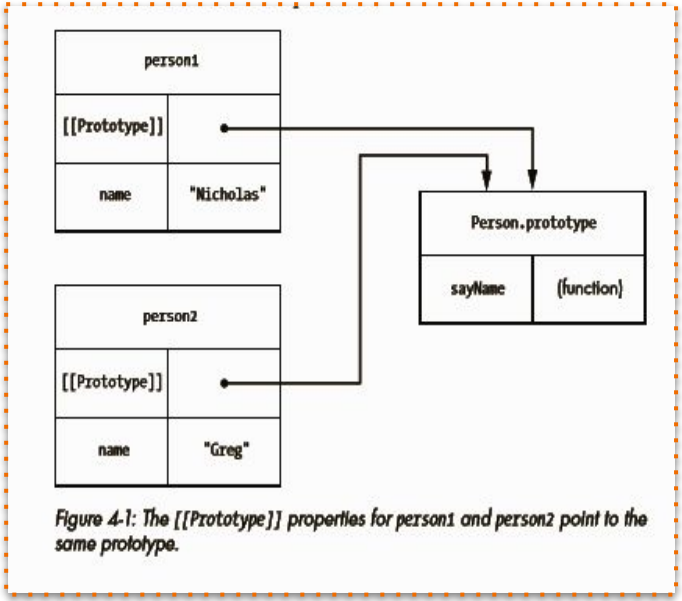

Figure 4-1: The [[Prototype]] properties for person1 and person2 point to the same prototype.

يمكنك قراءة قيمة الخاصية [[Prototype]] باستخدام Object. getPrototypeOf () على كائن .

على سبيل المثال ، يتحقق الكود التالي من [[Prototype]] لكائن عام generic وفارغ empty .

[[Prototype]] is always a reference to Object.prototype for any generic object like this one (1) .

```
1- var object = {};
   var prototype = Object.getPrototypeOf(object);
   console.log(prototype === Object.prototype);     // true
```

تدعم بعض محركات JavaScript أيضًا خاصية تسمى __proto__ على جميع الكائنات .
تسمح لك هذه الخاصية بالقراءة من والكتابة إلى خاصية [[Prototype]] .
يدعم كل من Firefox و Safari و Chrome و Node.js هذه الخاصية ، و __proto__ يسير على طريق التوحيد القياسي في ECMAScript 6.

You may also use the isPrototypeOf() function, which is included on all objects, to see if one object is a prototype for another :

```
var object = {};
console.log(Object.prototype.isPrototypeOf(object));    // true
```

Because object is a generic object , its prototype should be Object.prototype , meaning that isPrototypeOf() should return true.

* When the JavaScript engine reads a property from an object , it looks for an own property with the same name.
* If the engine finds an own property with the correct name, it returns that value.
* If the target object does not have an own property with that name, JavaScript checks the [[Prototype]] object instead.
* The value of a prototype property with that name is returned if one exists.
* Undefined is returned if the search fails to discover a property with the correct name.

ضع في اعتبارك ما يلي ، حيث يتم إنشاء كائن لأول مرة بدون أي خصائص خاصة به :

```
    var object = {};
1- console.log(object.toString());    // "[object Object]"
    object.toString = function() {
    return "[object Custom]";
};
2- console.log(object.toString());    // "[object Custom]"
    // delete own property
       delete object.toString;

3- console.log(object.toString());    // "[object Object]"
// no effect - delete only works on own properties
    delete object.toString;
    console.log(object.toString());    // "[object Object]"
```

* The toString() function in this example is created from the prototype and returns "[object Object]"1 by default.
* If you create an own property named toString(), it will be utilized every time toString() is used on the object 2.
* Only when the own property of the object 3 is removed is the prototype property used again.

(Keep in mind that the delete operator only works on own properties, so you can't delete a prototype property from an instance.)
Figure 4-2 explains what is going on in this case.

This example also explains an important concept:
A prototype property cannot be assigned a value from an instance.

As you can see in the midsection of Figure 4-2, giving toString a value generates a new own property on the instance while leaving the prototype property alone.

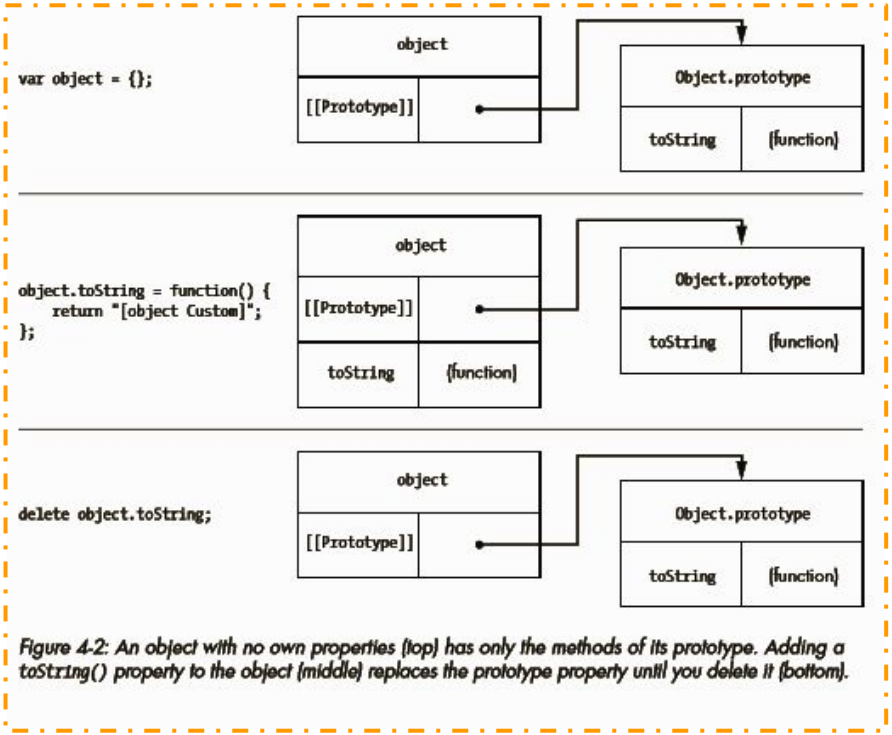

Figure 4-2: An object with no own properties (top) has only the methods of its prototype. Adding a toString() property to the object (middle) replaces the prototype property until you delete it (bottom).

Using Prototypes with Constructors

- **Prototypes are great for defining methods once for all objects of a particular type because of their common nature.**
- **There's no reason each instance has its own set of methods because methods tend to do the same thing for all instances.**
- **It's MUCH more efficient to put the methods on the prototype and then get the current instance through that.**
- **Consider the following new Person constructor as an example :**

* SayName() is declared on the prototype 1 rather than in the constructor in this version of the Person constructor.

* Even because sayName() is now a prototype property instead of an own property, the object instances work exactly the same as they did in the previous chapter's example.

* Other forms of data can be stored on the prototype as well, while reference values should be used with care.
=> You might not expect one instance to be able to update values that another instance would access because these data are shared across instances.

```
function Person(name) {
    this.name = name;
}
1-  Person.prototype.sayName = function() {
    console.log(this.name);
};
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
console.log(person1.name);     // "Nicholas"
console.log(person2.name);    // "Greg"
person1.sayName();      // outputs "Nicholas"
person2.sayName();     // outputs "Greg"
```

* This is what can happen if you don't pay attention to where your reference values are pointing:

```
function Person(name) {
    this.name = name;
}
Person.prototype.sayName = function() {
    console.log(this.name);
};
1-Person.prototype.favorites = [];
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
person1.favorites.push("pizza");
person2.favorites.push("quinoa");
console.log(person1.favorites);  // "pizza,quinoa"
console.log(person2.favorites);  // "pizza,quinoa"
```

Person1.favorites and person2.favorites both point to the same array because the favorites property (1) is defined on the prototype.
Any values you add to either person's favorites will become parts of the prototype's favorites array.
Because it could not be the behavior you want, it's necessary to be careful about what you declare on the prototype.

=> Despite the fact that you can add properties to the prototype one at a time, many developers choose to implement a more concise method that entails replacing the prototype with an object literal:

```
function Person(name) {
      this.name = name;
 }
Person.prototype = {
1-     sayName: function() {
          console.log(this.name);
       },
 2-   toString: function() {
         return "[Person " + this.name + "]" ;
      }
};
```

* On the prototype, this code specifies two methods : sayName() (1) and toString() (2).

Because it eliminates the need to type Person.prototype lots of times, this pattern has grown fairly popular.
However, there is one negative side effect to be careful of:

```
   var person1 = new Person("Nicholas");
   console.log(person1 instanceof Person);      // true
   console.log(person1.constructor === Person);   // false
1-console.log(person1.constructor === Object);   // true
```

Overwriting the prototype with object literal notation modified the constructor property to point to Object (1) instead of Person.
Because the constructor property is only on the prototype, not the object instance, this happened.
When a function is generated, its prototype property is set to the function's constructor property.
This pattern entirely overwrites the prototype object, meaning that the constructor will be taken from the newly constructed (generic) object supplied to Person.prototype.

To avoid this, while overwriting the prototype, set the constructor property to the correct value:

```
function Person(name) {
    this.name = name;
}
Person.prototype = {
1-  constructor: Person,
    sayName: function() {
    console.log(this.name);
    },
    toString: function() {
      return "[Person " + this.name + "]";
    }
};
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
console.log(person1 instanceof Person);         // true
console.log(person1.constructor === Person);    // true
console.log(person1.constructor === Object);    // false
console.log(person2 instanceof Person);         // true
console.log(person2.constructor === Person);    // true
console.log(person2.constructor === Object);    // false
```

The constructor property is expressly assigned to the prototype (1) in this case. The fact that there is no direct link between the constructor and the instance is perhaps the most exciting feature of the interactions between constructors, prototypes, and instances.

However, the instance and the prototype, as well as the prototype and the constructor, have a direct relationship.

This relationship is depicted in Figure 4-3.

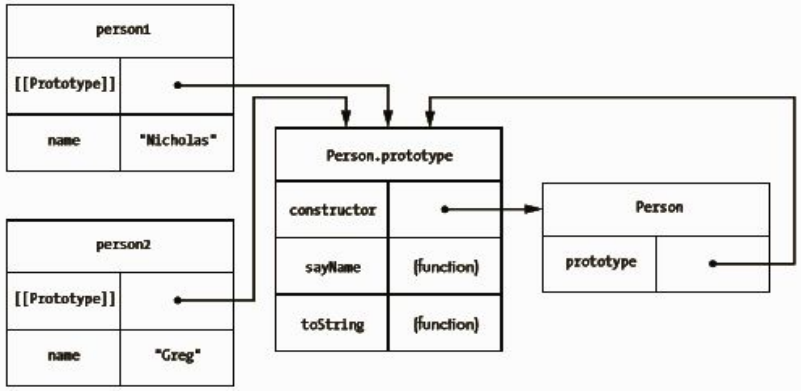

Figure 4-3: An instance and its constructor are linked via the prototype.

Changing Prototypes

* You can enhance all of those objects together at any moment since all instances of a certain type reference a shared prototype.
* Remember that the [[Prototype]] property only includes a pointer to the prototype, therefore any changes to the prototype will be immediately visible on any instance that references it.
* That means you can add new members to a prototype at any time and have those changes reflected on current instances, as shown in the following example:

```
function Person(name) {
    this.name = name;
}
Person.prototype = {
    constructor: Person,
1-  sayName: function() {
    console.log(this.name);
    },
2-  toString: function() {
      return "[Person " + this.name + "]";
    }
};
3- var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
console.log("sayHi" in person1);       // false
console.log("sayHi" in person2);       // false
// add a new method
4- Person.prototype.sayHi = function() {
   console.log("Hi");
};
 5- person1.sayHi();           // outputs "Hi"
    person2.sayHi();           // outputs "Hi"
```

The Person class has only two methods, sayName()1 and toString()2, in this code.
The sayHi()4 method is added to the prototype when two instances of Person are generated 3.
After that, both instances will be able to use sayHi() 5.
The search for a named property occurs every time that property is accessed, resulting in a smooth experience.
The capacity to make changes to the prototype at any time has some intriguing implications for sealed and frozen things.
When you call Object.seal() or Object.freeze() on an object, you're only affecting the instance and its own attributes.
On frozen objects, you can't add new own properties or update current own properties, but you can add properties to the prototype and continue expanding those objects, as seen in the following listing.

```
  var person1 = new Person("Nicholas");
  var person2 = new Person("Greg");
1-Object.freeze(person1);
2- Person.prototype.sayHi = function() {
     console.log("Hi"); };
    person1.sayHi();          // outputs "Hi"
    person2.sayHi();          // outputs "Hi"
```

There are two instances of Person in this example.
The first (person1) is a frozen 1, and the second (object) is a regular object.
Both person1 and person2 gain a new method when sayHi() is added to the prototype 2, seemingly contradicting person1's frozen condition.
The [[Prototype]] property is an instance's own property, and while the property is frozen, the value (an object) is not.

Built-in Object Prototypes

You might be thinking if prototypes allow you to modify the built-in objects that come standard with the JavaScript engine at this stage.
Yes, it is correct.
All built-in objects have constructors, which means you can change their prototypes. Modifying Array.prototype, for example, is all it takes to introduce a new method that can be used on all arrays.

On Array.prototype, this example introduces a function called sum() that simply adds all of the objects in the array and returns the result.
Through the prototype, the numbers array has automatic access to that method.
Because numbers is an instance of Array inside of sum(), the method is free to use other array methods like reduce ().
Strings, integers, and Booleans all have basic wrapper types that can be used to access primitive data as if they were objects, as you may recall.
You may really add extra functionality to the primitive values if you edit the primitive wrapper type prototype as in this example:

```
Array.prototype.sum = function() {
    return this.reduce(function(previous, current) {
      return previous + current;
    });
};
var numbers = [ 1, 2, 3, 4, 5, 6 ];
var result = numbers.sum();
console.log(result);              // 21
```

```
String.prototype.capitalize = function() {
    return this.charAt(0).toUpperCase() + this.substring(1);
};
var message = "hello world!";
console.log(message.capitalize());  // "Hello world!"
```

summary ^_^

- Constructors are simply regular functions that are invoked using the new operator.
- If you want to create many objects with the same characteristics, you can define your own constructors.
- Using instanceof or directly accessing the constructor property, you can identify objects generated by constructors.
- Any attributes shared among objects created with a particular constructor are defined by the prototype property of every function.
- Prototypes are often used to create shared methods and primitive value properties, while constructors are used to create all other attributes.
- Because the constructor attribute is shared all object instances, it is declared on the prototype.
- The [[Prototype]] property stores an object's prototype internally.
- This is a reference, not a duplicate, of the property.
- Because of the way JavaScript looks up properties, if you modify the prototype at any point in time, those changes will be reflected in all instances.
- When you try to access a property on an object, it looks for any own properties that have the name you specify.
- Because of this searching technique, the prototype can change at any time, and object instances that reference it will quickly reflect those changes.
- Prototypes for built-in items can also be updated.
- While doing so in production is not encouraged, it might be useful for testing and proofs of concept for new functionality.