مقرر الجافا سكريبت الباب الخامس (5) من كتاب
THE PRINCIPLES OF OBJECT ORIENTED JAVASCRIPT

الوراثة
Inheritance

اعداد الطالبة : غدي فوزي التمتام

تحت اشراف : أ. مصطفى قاباج

javascript

2021

المحاور

01 Prototype Chaining and Object.prototype

02 Constructor Inheritance

03 Constructor Stealing

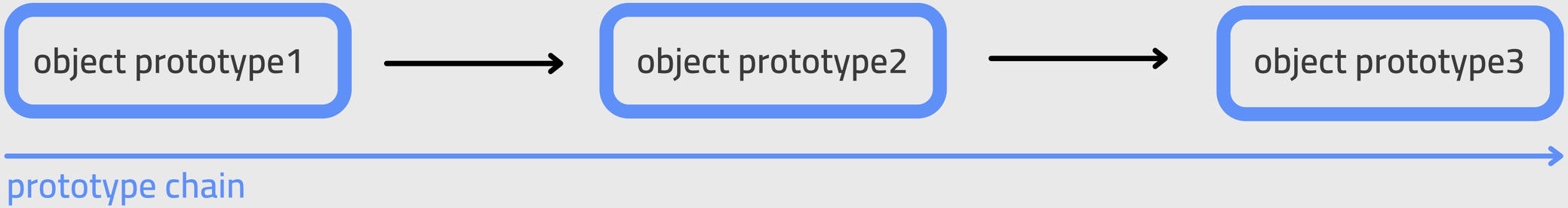
04 Accessing Supertype Methods

المقدمة

- تعلّم كيفية إنشاء الكائنات Object هي الخطوة الأولى لفهم البرمجة الموجهة نحو الكائنات OOP .
- الخطوة الثانية هي فهم عملية الوراثة Inheritance .
- في اللغات التقليدية الموجهة للكائنات , ترث الفئات Classes خصائص من فئات أخرى .
- أما في الجافا سكريبت , يمكن أن تحدث الوراثة Inheritance بين الكائنات Objects .

تسلسل النمـوذج الأولـي - Prototype Chaining

- هي عبارة عن سلسلة الـ prototype الخاص بأي كائن ، أي آن أي كائن في JavaScript لدية prototype وهذا الـ prototype هو في الأساس الأول عبارة عن كائن هو الأخر ، وبالتالي هو أيضا لديه prototype وهكذا تستمر السلسلة إلى أن تصل إلى الـ Master Object , وهذه السلسلة تسمي الـ prototype chain .



النموذج الكائنـي - Object.prototype

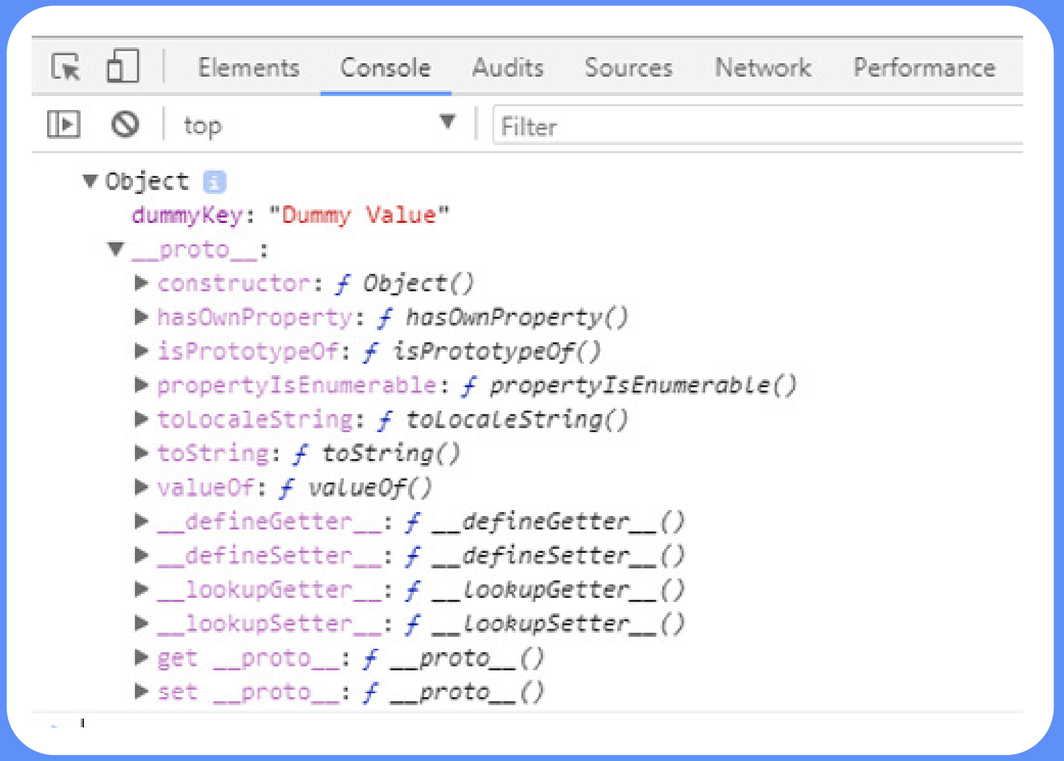
- أي كائن سوف تنشئه في JavaScript سوف يكون له نموذج مبدئي prototype .
ولك أن تعرف أن الـ prototype هذا هو في الأساس أيضا كائن object

على سبيل المثال

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

- يحتوي كائن الكتاب book هنا على نموذج أولي prototype يساوي الكائن النموذجي الأولي .
- وهذا هو السلوك الإفتراضي , حيث يكون لكل كائن عند انشائه نموذج أولي يحتوي على عمليات Method افتراضية .

العمليات Methods الافتراضية الموجودة في كائن النوذج الأولي Object.prototype



- hasOwnProperty()

يختبر إذا ماكانت الخاصية المكتوب اسمها موجودة أو لا

- propertyIsEnumerable()

يختبر إذا ماكانت الخاصية هي عدد لا يحى

- isPrototypeOf()

يختبر إذا ماكان هذا الكائن هو نموذج أولي لكائن أخر

- valueOf()

إرجاع قيمة تمثل الكائن

- toString()

إرجاع سلسلة تمثل الكائن

valueOf() 1

تستدعي JavaScript طريقة valueOf لتحويل كائن إلى قيمة بدائية بشكل افتراضي ،
و إذا كان الكائن ليس له قيمة بدائية , valueOf تقوم بإرجاع الكائن نفسه

على سبيل المثال :

```
1  var now = new Date() ;
2   var earlier = new Date(2010, 1, 1);
3   console.log(now > earlier);  // true
```

في هذا المثال دالة الـ valueOf() تستدعى عند كلا
الكائنين 1 & 2 قبل إجراء عملية المقارنة

toString() 2

تستدعي JavaScript طريقة toString() لإرجاع سلسلة تمثل الكائن ,
و يحتوي كل كائن على طريقة toString() التي يتم استدعاؤها تلقائيا عندما يتم تمثيل الكائن
كقيمة نصية أو عندما يتم الإشارة إلى كائن بطريقة يتوقع بها سلسلة بشكل افتراضي .

على سبيل المثال :

```
1  var book = {
2   title: "The Principles of Object-Oriented JavaScript"
3  };
4   var message = "Book = " + book;
5  console.log(message); // "Book = [object Object]"
```

في هذا المثال عند استدعاء الكائن في 4 قام البرنامج باستدعاء دالة toString()
الموروثة من Object.prototype لتعطي القيمة الافتراضية لها وهي [object Object]

دعونا نجرب ان نعدل في الدالة toString ()

```
var book = {
 title: "The Principles of Object-Oriented JavaScript"
toString: function() {
 return "[Book " + this.title + "]"
} };
 var message = "Book = " + book;
 console.log(message);
// "Book = [Book The Principles of Object-Oriented JavaScript]"
```

تعديل النموذج الأولي للكائن
Modifying Object.prototype

كما عرفنا انه كل الكائنات لديها كائن ترث منه بشكل تلقائي ويسمي Object.prototype
وهذا الكائن يوثر على كل الكائنات ومن الخطير التعديل عليه

```
Object.prototype.add = function(value) {
 return this + value;
 };
var book = {
title: "The Principles of Object-Oriented JavaScript"
 };
console.log(book.add(5)) ; // "[object Object]5"
console.log("title".add("end")) ; // "titleend"

// in a web browser
console.log(document.add(true)); // "[object HTMLDocument]true"
console.log(window.add(5)) ; // "[object Window]true"
```

لنرى ماسيحدث إذا
قمنا بالتعديل عليه !

عند إضافة دالة add() لـ النموذج الأولي
Object.prototype سوف يكون لكل
الكائنات نموذج أولي موروث به هذه
الدالة سواء كان ذلك منطقيا أم لا !

جانب أخر من المشكلة

خاصية الـ add() في المثال السابق هي خاصية لا حصر لها , لذلك ستسبب مشاكل عند استخدام for-in loop كما في المثال الآتي :

```
var empty = {};
 for (var property in empty)
 {
console.log(property);
 }
```

لان التعديل لديه القدرة على التأثير على الكثير من التعليمات البرمجية يُنصح بإستخدام hasOwnProperty() في عملية الـ for-in loop

```
var empty = {};
 for (var property in empty)
 {
if (empty.hasOwnProperty(property)) {
console.log(property); }
 }
```

Object Inheritance
وراثة الكائنات

طريقـة Object.create()

```
var book = Object.create(Object.prototype, {
title: {
configurable: true,
enumerable: true,
value: "The Principles of Object-Oriented
JavaScript",
writable: true
} });
```

تأخذ هذه الطريقة الإعلان عن الكائن بنفس طريقة الـ Literal ولكن تكون بشكل صريح عن طريق الـ Object.create()

طريقـة Literal

```
var book = {
title: "The Principles of Object-Oriented JavaScript"
};
```

هذا الكائن يرث تلقائيا من نموذج الكائن Object.prototype ويتم تعيين الخصائص(configurable, enumerabl , writable) بشكل افتراضي والدوال المذكورة سابقا أيضاً , وعند اسناد دالة للكائن يقوم البرنامج أولا بالبحث عنها داخل الكائن نفسه وان لم يجدها سيبحث في الـ Prototype الخاص بها الذي سيكون افتراضي .

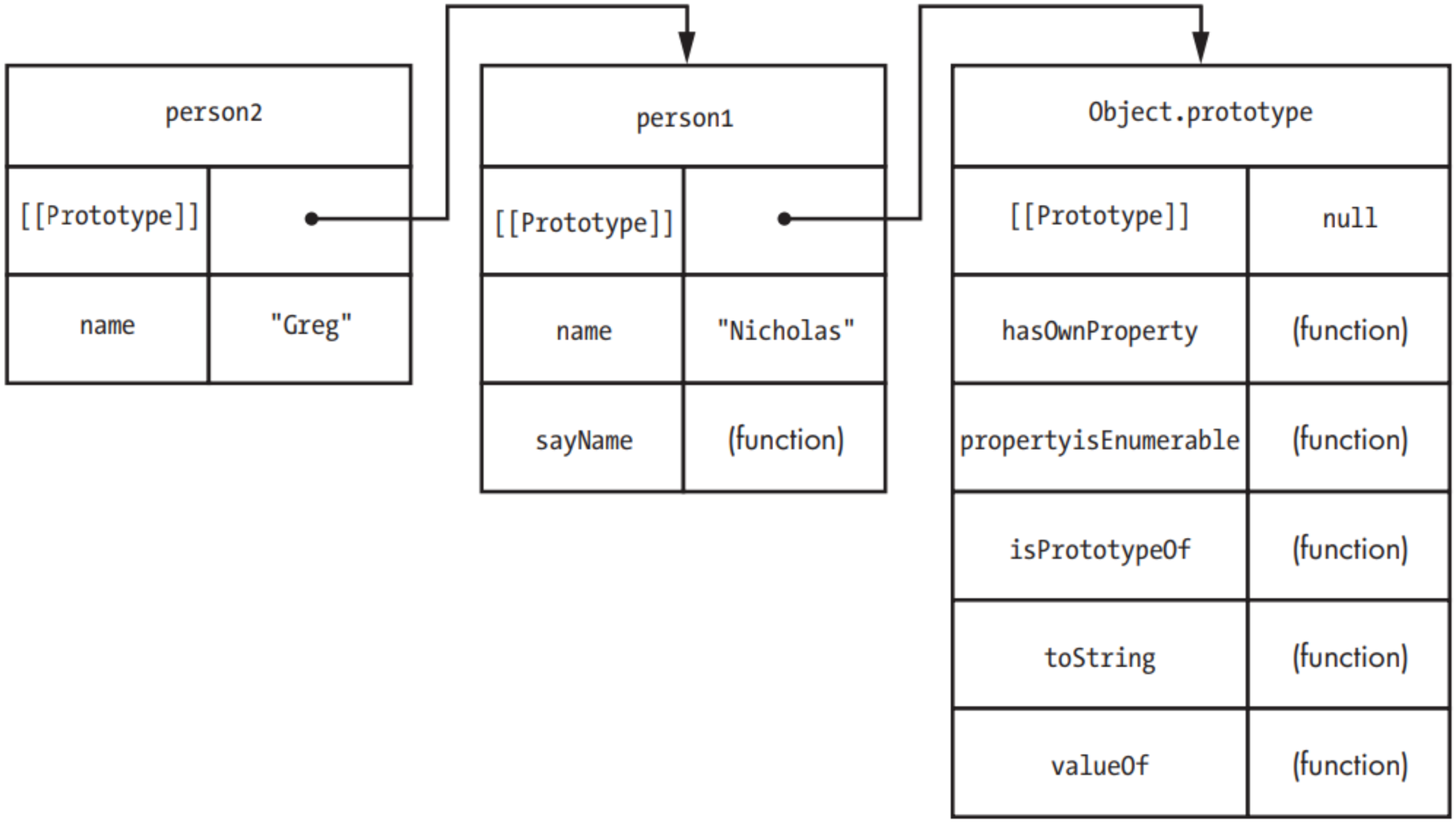
وراثة الكائن - Object Inheritance

في هذا المثال قمنا بإنشاء كائن جديد واسميناه person2 ويقوم هذا الكائن بوراثة خصائص وأفعال الـ person1 عن طريق تحديده كـ Prototype مكان الافتراضي وتم ذلك عن طريق الـ Object.create

ومن ثمّ قمنا بإختبار مقارنة بإستخدام دوال الـ Prototype الإفتراضية لنرى كيف يجري الأمر !

```
var person1 = {
name: "Nicholas",
 sayName: function() {
console.log(this.name);
} };
var person2 = Object.create(person1,
 {
name: {
configurable: true,
enumerable: true,
 value: "Greg",
writable: true
 } });
 person1.sayName(); // outputs "Nicholas"
person2.sayName(); // outputs "Greg"
 console.log(person1.hasOwnProperty("sayName")); // true
console.log(person1.isPrototypeOf(person2)); // true
console.log(person2.hasOwnProperty("sayName")); // false
```

prototype chains



وأيضا يمكنك انشاء كائن من غير ان يحتوي على Prototype

إليك هذا المثال

```
var nakedObject = Object.create(null);
 console.log("toString" in nakedObject); // false
console.log("valueOf" in nakedObject); // false
```

الكائن nakedObject في هذا المثال هو كائن بدون سلسلة نموذج أولي prototype chain .
وهذا يعني أن الدوال الموجودة في الـ Prototype مثل toString() و valueOf() غير موجودة على الكائن.
في الواقع , هذا الكائن عبارة عن قائمة فارغة تماما بدون خصائص محددة مسبقا , مما يجعله مثاليا لإنشاء تجزئة بحث دون تصادمات تسمية محتملة مع أسماء الخصائص الموروثة. لا توجد العديد من الاستخدامات الأخرى لكائن مثل هذا , ولا يمكنك استخدامه كما لو كان يرث من كائن النموذج الأولي .

Constructor Inheritance

كل وظيفة function تقريبا لديها خاصية النموذج الأولي التي يمكن تعديلها أو استبدالها. يتم تعيين خاصية النموذج الأولي prototype property تلقائيا لتكون كائن Objectعام جديد يرث من كائن النموذج الأوليObject.prototype ولها خاصية خاصة واحدة تسمى Constructor .

```
// you write this
function YourConstructor() {
 // initialization
}
// JavaScript engine does this for you behind the scenes
YourConstructor.prototype = Object.create(Object.prototype, {
 constructor: {
 configurable: true,
 enumerable: true,
 value: YourConstructor
 writable: true
 }
 });
```

يقوم محرك JavaScript بما يلي
نيابة عنك خلف الكواليس

لنتمعنّ في هذا المثـال !

```
function Rectangle(length, width) {
 this.length = length;
 this.width = width;
}
Rectangle.prototype.getArea = function() {
 return this.length * this.width;
};
Rectangle.prototype.toString = function() {
return "[Rectangle " + this.length + "x" +
this.width + "]";
};
// inherits from Rectangle
 function Square(size) {
 this.length = size;
 this.width = size;
}
```

```
Square.prototype = new Rectangle();
Square.prototype.constructor = Square;
Square.prototype.toString = function() {
 return "[Square " + this.length + "x" + this.width + "]";
};
var rect = new Rectangle(5, 10);
var square = new Square(6);
```

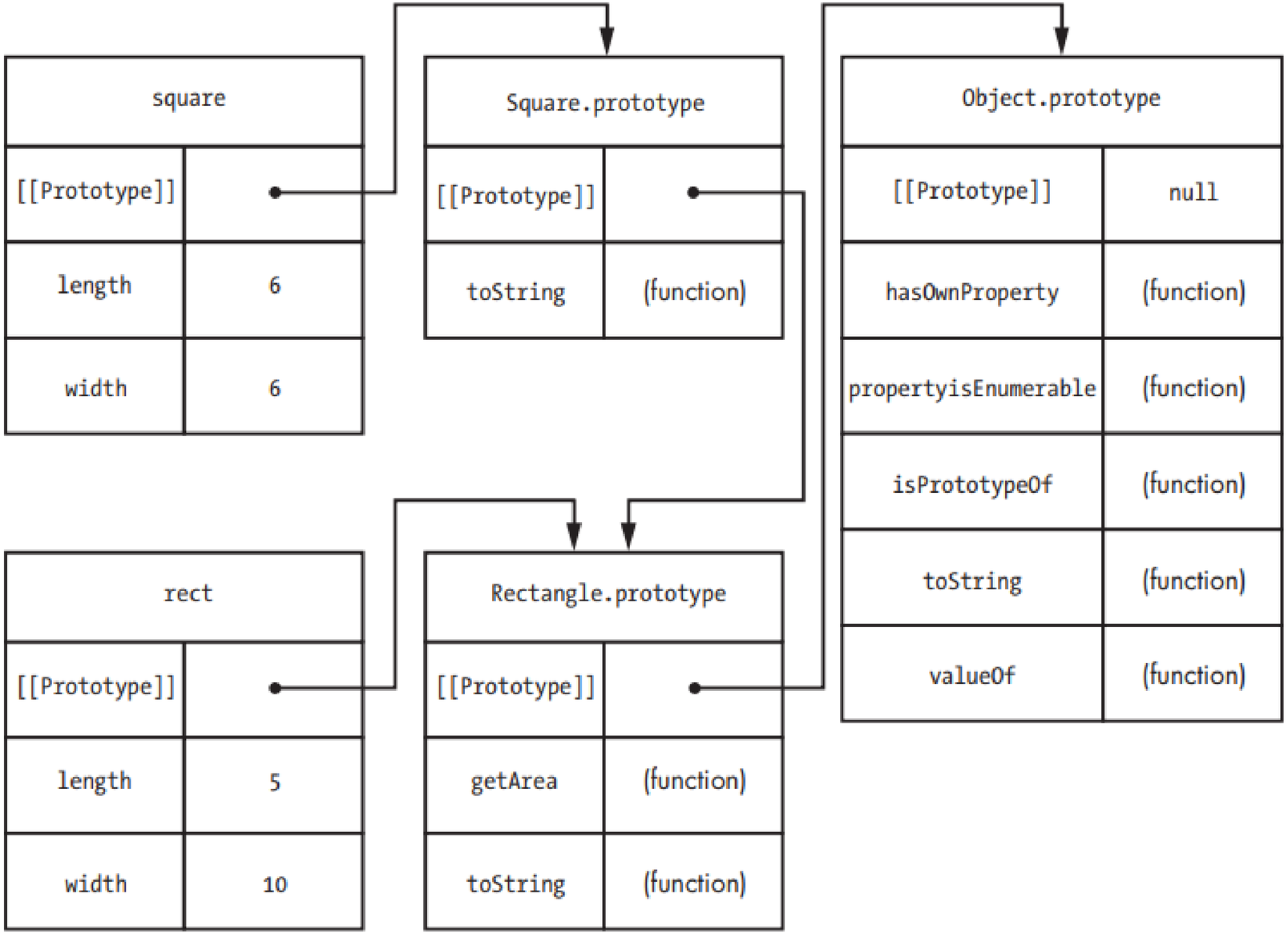
ماذا حدث في المثال السّابق يا ترى !

تم انشاء دالة للتعبير عن المستطيل بإسم Rectangle وتم اعطائها خصائص للطول والعرض وانشاء دالة داخلها ترجع قيمة مساحة المثلث getArea() , واستبدال الداله الموجودة في الـ prototybe الخاص بدالة المستطيل باسم ToString() ترجع هذا الشكل [Rectangle length x width] , وتم انشاء دالة آخرى للتعبير عن المربع بإسم Square وإعطائه خصائص الطول والعرض , وتم جعل الـPrototype الخاص به هو كائن الـRectangle يعني كائن المربع سيرث خصائص وافعال كائن المستطيل , وتم استبدال أيضا دالة اخرى في كائن الـ Prototype الخاص بالمربع بإسم ToString() ترجع هذا الشكل [Square length x width] , وتم انشاء Object بإسم rac يورث من الـ Rectangle
, و Object بإسم squareيورث من الـ Square ,

```
console.log(rect.getArea()); // 50
console.log(square.getArea()); // 36
console.log(rect.toString()); // "[Rectangle 5x10]"
console.log(square.toString()); // "[Square 6x6]"
console.log(rect instanceof Rectangle); // true
console.log(rect instanceof Object); // true
console.log(square instanceof Square); // true
console.log(square instanceof Rectangle); // true
console.log(square instanceof Object); // true
```

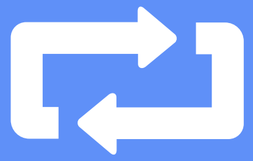
لنجري الأن بعض العمليات على هذه الكائنات ونرى ماذا سيحدث !

prototype chains



سنعيد المثال السابق وعملية الوراثة , ولكن هذه المرة باستخدام Object.create()

```
// inherits from Rectangle
function Square(size) {
 this.length = size;
 this.width = size;
}
Square.prototype = Object.create(Rectangle.prototype, {
 constructor: {
 configurable: true,
 enumerable: true,
 value: Square,
 writable: true
 }
 });
Square.prototype.toString = function() {
 return "[Square " + this.length + "x" + this.width + "]";
};
```



في هذا الكود
الكائن Square.prototype سيقوم بالوراثة من Rectangle.prototype عن طريق الـ Object.create()
هذا يعني انه لا داعي للقلق بشأن التسبب في الأخطاء عن طريق استدعاء constructor بدون البرامترات بعد الآن .
خلاف ذلك , يتصرف هذا الكود تماما مثل الكود السابق , تظل سلسلة النموذج prototype chains الأولي سليمة .

ملاحظة : تأكد دائما من استبدال النموذج الأولي prototype قبل إضافة خصائص إليه , أو ستفقد العمليات methods المضافة عند حدوث الـoverwrite.

Constructor Stealing

```
function Rectangle(length, width) {
this.length = length;
this.width = width; }
Rectangle.prototype.getArea = function() {
return this.length * this.width; };
Rectangle.prototype.toString = function() {
 return "[Rectangle " + this.length + "x" + this.width + "]"; };
// inherits from Rectangle
function Square(size) {
Rectangle.call(this, size, size);
// optional: add new properties or override existing ones here
}
Square.prototype = Object.create(Rectangle.prototype, {
constructor: {
 configurable: true,
 enumerable: true,
 value: Square,
writable: true } });
```

```
Square.prototype.toString = function() { return "[Square " +
this.length + "x" + this.width + "]"; };
var square = new Square(6);
 console.log(square.length); // 6
console.log(square.width); // 6
 console.log(square.getArea()); // 36
```

نظرا لأنه يتم تحقيق الوراثة من خلال سلاسل النموذج الأولي prototype chains في JavaScript , فلن تحتاج إلى استدعاء constructor supertype الخاص بالكائن.
و إذا كنت ترغب في استدعاء constructor من supertype , constructor subtype فأنت بحاجة إلى الاستفادة من كيفية عمل وظائف JavaScript.

شرح لما يحدث في الكود السابق !

- الـ constructor الخاص بالمربع يستدعي الـ constructor الخاص بالمستطيل , ويقوم بتمرير الـsize مرتين (مرة واحدة للطول length ومرة واحدة للعرضwidth). يؤدي القيام بذلك إلى إنشاء خصائص الطول والعرض على الكائن الجديد ويجعل كل منها مساويا للحجم
- هذه هي الطريقة ! لتجنب إعادة تعريف الخصائص من constructor الذي تريد أن ترث منه.
- يمكنك إضافة خصائص جديدة أو تجاوز الخصائص الموجودة بعد تطبيق supertype constructor .
- هذه العملية المكونة من خطوتين مفيدة عندما تحتاج إلى تحقيق عملية الوراثة بين أنواع معينة .
- ستحتاج دائما إلى تعديل الـ constructor’s prototype , وقد تحتاج أيضا إلى استدعاء supertype constructor من داخل subtype constructor .
- عموما , عليك تعديل النموذج الأولي prototype لعملية الوراثة واستخدام خصائص constructor stealing .

Accessing Supertype Methods

- في المثال السّابق ، يحتوي النوع المربع على دالة الـ toString() الخاصة به والتي تظلل الـ tostring() الخاصة بالنموذج الأولي prototype .
- من الشائع إلى حد ما تجاوز طرق supertype بوظائف جديدة من النوع subtype ، ولكن ماذا لو كنت لا تزال ترغب في الوصول إلى طريقة supertype ؟ بلغات أخرى ، قد تكون قادرا على قول super.toString() ، ولكن الجافا سكريبت ليس لديها أي شيء مماثل.
- فبدلا من ذلك ، أصبح يمكنك الوصول مباشرة إلى supertype’s prototype واستخدام إما call() أو apply() لتنفيذ العملية على كائن subtype .

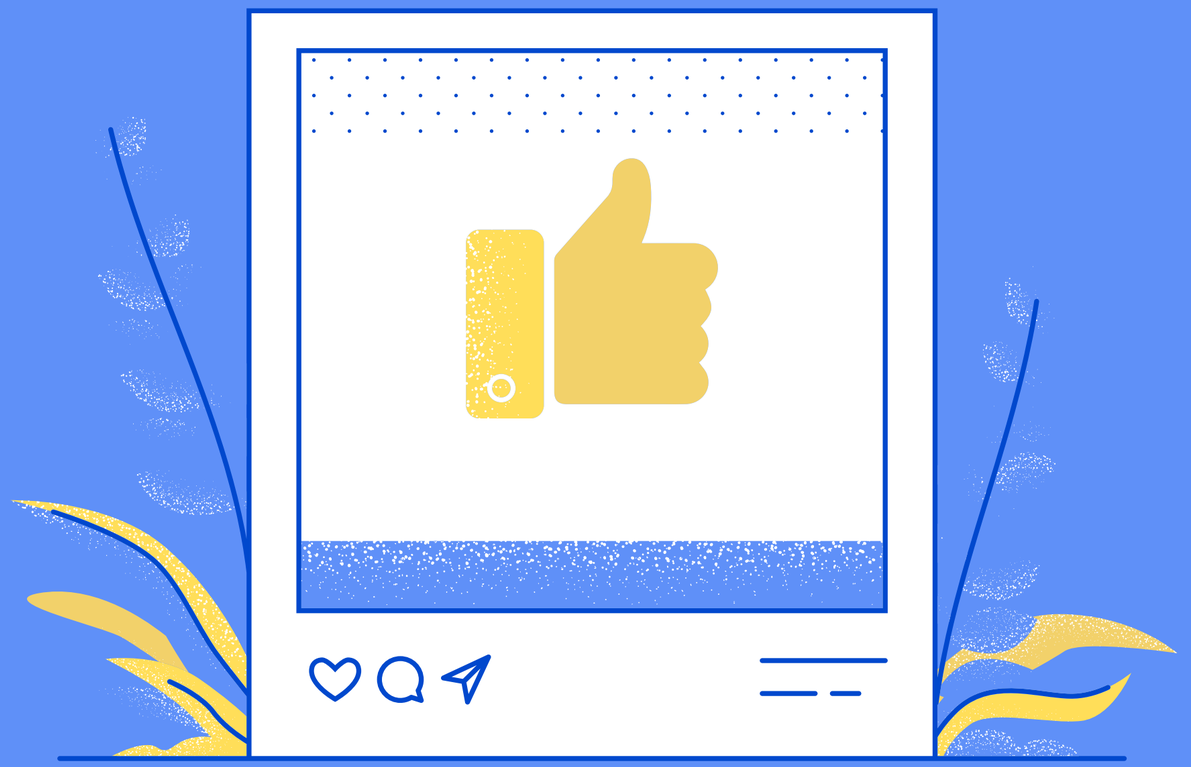
على سبيل المثال

```
function Rectangle(length, width) {
 this.length = length;
 this.width = width;
}
Rectangle.prototype.getArea = function() {
 return this.length * this.width  };
Rectangle.prototype.toString = function() {
return "[Rectangle" + this.length + "x" + this.height + "]"};
// inherits from Rectangle
function Square(size) {
 Rectangle.call(this, size, size) };
Square.prototype = Object.create(Rectangle.prototype, {
 constructor: {
 configurable: true,
 enumerable: true,
 value: Square,
 writable: true }  });
// call the supertype method
Square.prototype.toString = function() {
 var text = Rectangle.prototype.toString.call(this);
 return text.replace("Rectangle", "Square") };
```

الملخص

- جافا سكريبت تدعم الوراثة من خلال تسلسل النموذج prototype chain , حيث يتم إنشاء سلسلة النموذج الأولي بين الكائنات Objects عندما يتم تعيين [[Prototype]] من كائن ما وجعله يساوي الآخر.
- جميع الكائنات العامة Objects ترث تلقائيا من كائن النموذج الأولي prototype .
- إذا كنت ترغب في إنشاء كائن يرث من كائن آخر غير الإفتراضي , يمكنك استخدام Object.create() لتحديد قيمة [[Prototype]] للكائن جديد.
- يمكنك تحقيق الوراثة بين أنواع معينة عن طريق إنشاء prototype chain على constructor .
- عن طريق تعيين خاصية الـ constructor’s prototype إلى قيمة أخرى , سيمكنك إنشاء عملية وراثة بين حالات من النوع المخصص و كائن النموذج الأولي prototype لتلك القيمة الأخرى .
- تشترك جميع حالات هذا constructor في نفس النموذج الأولي prototype , لذلك ترث جميعها من نفس الكائن .
- لوراثة الخصائص الخاصة بشكل صحيح , يمكنك استخدام constructor stealing , والتي تقوم ببساطة باستدعاء وظيفة constructor باستخدام call() أو apply() بحيث يتم إجراء أي تهيئة على كائن النوع الفرعي subtype .
- الجمع بين constructor stealing و تسلسل النموذج prototype chaining هو الطريقة الأكثر شيوعا لتحقيق عملية الوراثة بين أنواع مخصصة في جافا سكريبت JavaScript .
- وغالبا ما يسمى هذا المزيج بـ inheritance pseudoclassical بسبب تشابهه مع عملية الوراثة في اللغات المبنية على الطبقات .
- يمكنك الوصول إلى الطرق على supertype عن طريق الوصول مباشرة إلى النموذج الأولي supertype. عند القيام بذلك , يجب عليك استخدام call() أو apply() لتنفيذ طريقة supertype على كائن النوع الفرعي subtype .

الوراثة Inheritance



مقرر الجافا سكريبت الباب الخامس (5) من كتاب
THE PRINCIPLES OF OBJECT ORIENTED JAVASCRIPT

اعداد الطالبة : غدي فوزي التمتام

تحت اشراف : أ. مصطفى قاباج

javascript

2021