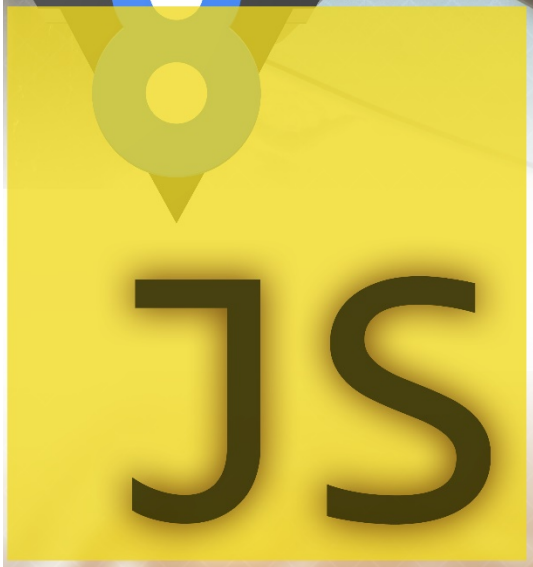
جافا سكريبت تحت الغطاء

دليلك نحو فهم أعمق للغة و صناعة مكتبتك الخاصة



- نظرة على مفاهيم متقدمة
- نظرة على شيفرة jQuery
- إنشاء مكتبة بسيطة

إعداد و تأليف:

حديدي سفيان

الفهرس

حول المؤلف:

حديدي سفيان هو شاب جزائري خريج جامعة بجاية بشهادة ليسانس تخصص إستغلال منجمي. يعمل في المجال الحر. يعشق البرمجة خصوصا الجافا سكريبت و البايثون، و علوم الحاسوب، علوم الكهرباء و الإلكترونيات، كل ما يتعلق بالتقنية و التكنولوجية. مالك لمدونة أوبنتو العرب.

يمكنك متابعتي على:

فيسبوك: **سفيان ولد رقان**

إنستغرام: **soufyane_weld_reggane**

تويتر: **سفيان ولد رقان**

و لمراسلتي عبر الإيميل: hesoka05@gmail.com.

شكر و عرفان:

أتقدم بخالص الشكر و العرفان لوالدي الكريمين اللذان ربّياني و علّماني، إلى أخي الحبيب الذي ساندني ماديا و معنويا، إلى أصدقائي الأعزاء أين ما كنتم. شكرا لكل من ساهم بقول أو فعل. جعل الله ذلك في ميزان حسناتكم.

شروط الترخيص:

إعترافات:

تصميم الغلاف: حديدي سفيان.
الصور في الشروحات: حديدي سفيان.
حفظا لحقوق النسخ، أشكر كل من aopsan و jcomp من موقع Freepik على الصور المستعملة في الغلاف و كما هو مطلوب في الرخصة وجب نسب أعمالهما إليهما:

""Designed by: aopsan / Freepik

"Designed by jcomp / Freepik"

مقدمة:

«و قل ربي زدني علما»

(طه - 114)

الحمد لله الذي علَّمَ بالقلم، علَّم الإنسانَ ما لم يعلَم، والصَّلاة والسَّلام على النبيِّ الأكرَم، نبيِّنا ومعلِّمنا محمد، صلَّى الله عليه وعلى آله وصحبه وسلَّم.

تعد لغة الجافا سكريبت، (الغامضة و الغريبة أو السهلة الممتنعة)، من أشهر اللغات البرمجية حاليا. ربما تكون مبتدئاً فيها، و ربما تستعملها منذ أعوام. و قد لا يخفى على الجميع أن أشهر المكتبات، و أطر العامل المعمول بها في الويب، و غيره، مكتوب بهذه اللغة الرائعة! فالسؤال الذي يطرح نفسه هنا: ما مدى تمكنك منها؟ و هل تتقن إستخدام مميزاتها؟ هل تستغل بعض الحيل فيها لأداء مبتغاك؟

حسنا هذا هو الغرض من هذا الكتاب، الصعود بك إلى مستوى الإحتراف في هذه اللغة، بل حتى صناعة المكتبات، و أطر العمل كتلك المشهورة حاليا! بحيث يأخذك في جولة داخل دهاليز اللغة، و يكشف لك ما يحدث خلف الكواليس. سيغوص بك في مفاهيم متقدمة، لتصبح بعدها قادرا على فهم ما يفهمه من صنع أشهر المكتبات كمكتبة jQuery مثلا! مما يجعلك أفضل كمطور ويب أو حتى مطور تطبيقات node.js و غيرها... فأي مبرمج لا يريد أن يصبح محترفا؟!

و لكن لنتفق في البداية على ما يلي:

يفترض في هذا الكتاب أنك على علم بأساسيات اللغة من المتغيرات و الدوال...الخ.

لن يتطرق الكتاب إلى صناعة صفحات تفاعلية أو تعليم مكتبة jQuery، بل سيتم التوغل في شيفرتها المصدرية و تفسير بعض السطور البرمجية لما له علاقة بالمفاهيم المتقدمة التي هي محور الكتاب.

هذا، و الله ولي التوفيق، و هو من وراء القصد.

الجزائر – أدرار في 2019\08\02 **المؤلف.**

نظرة على مفاهيم متقدمة

المحلل اللغوي:

المحلل اللغوي، أو محلل بناء الجملة هو: برنامج يقرأ الشيفرة الخاصة بك، و يحدد ما الذي تقوم به، و ما إذا كانت القواعد صحيحة أم لا.

فعند كتابتك لشيفرة جافا سكريبت، فإن الحاسوب لن يفهمها مباشرة، بل هناك برنامج يترجمها إلى اللغة التي يفهمها، يدعى هذا البرنامج بالمترجم أو المفسر.

تقوم المفسرات أو المترجمات بقراءة شيفرتك حرفا بحرف، و تحدد ما إذا كان البناء صحيحا (أي أن القواعد صحيحة)، و بعدها تنفذ هذا البناء، أو هاته القواعد في شكل يفهمه الحاسوب. مثلا لو كتبت:

```
function hello(){
var a = “hello world!”;
}
```

سيقوم المحلل اللغوي، أو المترجم بقراءتها حرفا، حرفا أي: f u n c t i o n فيعرف أنها الكلمة المفتاحية function (تدعى function بعد هذه العملية بالرمز أو المعلمة token)، و يجب أن يكون ورائها فراغ (مسافة)، و هكذا مع hello حيث يقرأها h e l l o حتى يصادف قوس الفتح (فيعرف أن hello إسم الدالة، و هكذا مع بقية الشيفرة. و ستترجم الشيفرة حسب النظام الذي وفره صانع اللغة؛ (تدعى هاته العملية بالتكوين أو الترميز tokenization). إذن ما سيتم تقديمه إلى الحاسوب ليست شيفرتك نفسها، بل ترجمة لها. هذا يعني أن خلال عملية الترجمة هاته، قد يقرر المترجم أن يقوم بأشياء أخرى حسب ما حدده له صانع اللغة (إذا أراد ذلك فعلا). و هذا أمر في غاية الأهمية؛ لفهم جافا سكريبت.

بإختصار: الشيفرة الخاصة بك ليست ما يتم تقديمه بالفعل للحاسوب، بل ترجمة عنها.

حسنا، قلنا أن المحلل يقرأ الشيفرة ليتأكد من صحة قواعدها، و ما تقوم به، لنفصل قليلا في هذا:

يعمل المحلل على التوقّع، و الإحتمالات. حيث ما أن يقرأ الحرف الأول، يبدأ بتوقّع الإحتمالات الممكنة، فبالنسبة للشيفرة السابقة، عند مروره على الحرف f، يتوقع أن الكلمة مفتاحية إما for، أو function، و بذلك ينتقل للحرف الثاني كي يحدد ماهية الكلمة، فإن وجد حرف o، توقعها الكلمة المفتاحية for، و إن وجد الحرف u، توقعها function.

حسنا، بعدما وجد أنها الكلمة المفتاحية function، هنا يأتي للنظر إلى القاعد المتعلقة بهاته الكلمة، و التي تنص على أنه يجب أن تتبع بمسافة، ثم إسم للدالة، ثم قوسي الفتح، و الإغلاق،

و بينهما يمكن أن تكون وسائط (parameters)، أو لا تكون. و بعد قوسي الوسائط -إن صح التعبير- لابد أن يأتي قوسي جسم الدالة، و هما القوسين المعقوفين { }، و بينهما يمكن أن تكون هناك شيفرة، و ربما لا. فيبدأ بالتحقق من هاته القواعد. و بالطبع هذا بالمرور حرفا بحرف لمعرفة ماهية كل كلمة، فإن كان كل شيئ حسب القواعد، فإنّ الشيفرة ستكون صحيحة "لغويا" بغض النظر عما تقوم به.

لكن ما إن يصادف أمرا مخالفا للقواعد، فإنه يوقف البرنامج في تلك النقطة، و يرجع خطأ دون أن يهتم لباقي الكود.

و بالمثال يتضح المقال:

```
function(){
    console.log("I'm anonymous!")
}
console.log("I'm right syntactically!");
```

```
Uncaught SyntaxError: Unexpected token (
```

في هذا المثال لم نضع إسما للدالة، و الذي كان يتوقع المحلل أن يجده بعد الكلمة المفتاحية function بمسافة طبعا، فتصادف مع قوس الفتح (و هذا مخالف للقواعد طبعا في هذه الحالة). نعلم أن الدوال المجهولة مسموح بها في جافا سكريبت لكن ليس بهذه الطريقة. بعد أن وجد المحلل أمرا غير مُتوقّع أوقف البرنامج على الفور، و تجاهل بقية السطور، رغم صحتها من حيث القواعد، و أرجع لنا خطأ يخبرنا ما المشكلة. لاحظ أنه يحدد نوع الخطأ ثم بعده رسالة الخطأ. حيث أن نوع الخطأ هنا بنيوي (التنسيق) SyntaxError أما الرسالة فهي: "ترميز غير متوقع" (Unexpected token)

ما عليك إدراكه هو أن بعض الأمور إختيارية، و بعضها إجباري من حيث البنية، فمثلا بالنسبة للوسائط (parameters) المتعلقة بالدوال، فتعريفها ليس إجباري، الأمر يعتمد عليك أنت، حيث أنت تعرف متى تحتاجها، و متى تستغني عنها. و كذلك بالنسبة لتمريرها في بعض الأحيان عند إستدعاء دوال تتوقع تمرير وسائط لها، ما يحدث فقط، هو نتائج غير متوقعة و غير مقبولة أحيانا. تابع الفصول القادمة لتفهم ما أقصد بالنسبة لهاته الأخيرة.

البيئة المعجمية:

في لغات البرمجة حيث البنية المعجمية مهمة؛ حيثما ترى الأشياء مكتوبة، يعطيك هذا فكرة عن مكان وضعها فعليا في ذاكرة الحاسوب، و كيف ستتفاعل مع عناصر البرنامج الأخرى من دوال و متغيرات، و هذا بسبب أن البرنامج الذي يحول شيفرتك إلى تعليمات ينفذها الحاسوب، يهتم بمكان وضعك للأشياء.

فعندما نتكلم عن البنية المعجمية لشيفرة ما فنحن نتكلم عن المكان التي كتبت فيه و ماذا يحيط بها!

سياق التنفيذ و البيئة العامة:

سياق التنفيذ:

هو غلاف يساعد في إدارة الشيفرة التي تشتغل.

هناك العديد من البيئات المعجمية. أيّ واحدة منها تنفذ الآن، تُدار بواسطة سياق التنفيذ.
يمكنها أن تحتوي على أشياء بعيدة عن ماذا كتبت في شيفرتك.

```
1  b();
2  console.log(a)
3
4  var a = 'Hey! how are?';
5  function b() {
6      console.log("be is called")
7  }
```

عند تنفيذ شيفرة جافا سكريبت؛ فإن محرك جافا سكريبت يقوم بإنشاء سياق تنفيذ، حيث يغطي أو يغلف هذا السياق تلك الشيفرة، و بالإضافة إلى هذا، يوفر لنا ثلاثة أشياء هي:

الكائن العام window، المتغير this، البيئة الخارجية.

و لو نظرنا إلى المتغير this، فإننا نجده يشير إلى الكائن العام window.

يشير الكائن العام إلى سياق التنفيذ الذي نتعامل معه (قد يشير إلى المستعرض إذا كنا نتعامل مع المستعرض، و قد يشير إلى كائن البيئة -التي نتعامل معها- كالـ node.js مثلا، و هكذا...)

يتم إنشاء سياق التنفيذ على مرحلتين:

1. **مرحلة الإنشاء.**

2. مرحلة التنفيذ.

مرحلة الإنشاء:

يتم في هذه المرحلة إنشاء المساحة المخصصة -في الذاكرة- لكل المتغيرات، و الدوال التي في الشيفرة، و يتم وضع الكائن العام window و المتغير this في الذاكرة أيضا.

خلال هذه المرحلة، يمر المحلل اللغوي عبر الشيفرة، و يبدأ في إعداد ما كتبته لتتم ترجمته، و تنفيذه. إذا فهو يعرف أين أُنشأت المتغيرات، و الدوال أيضا. بالإضافة إلى أنه يحجز لها المساحة على الذاكرة؛ تدعى هاته الخطوة بالرفع (hoisting).

و لهذا قبل أن يتم تنفيذ الشيفرة سطراً بسطر في مرحلة التنفيذ، فإن المحرك يستطيع الوصول إلى هاته المتغيرات.

كتوضيح، فإن الدوال توضع بكاملها في المساحة المخصصة لها في الذاكرة، إسمها، و الشيفرة التي بداخلها.

أما بالنسبة للمتغيرات، فإن محرك جافا سكريبت يخصص لها المساحة، و يضع لها القيمة الإبتدائية undefined، هذه القيمة تنوب مؤقتا عن القيمة المحددة أو المسندة إلى المتغير.

و معنى القيمة undefined هو: غير محدد. و هي نفس القيمة التي سنحصل عليها إذا لم نضع، أو نحدد أي قيمة للمتغير. سيتم وضع القيم الحقيقية في مرحلة التنفيذ، و التي سنتكلم عنها لاحقا.

تجدر الملاحظة هنا أن undefined ليست سلسلة نصية (string)، بل تعد قيمة خاصة محجوزة سابقا في اللغة. ولتتأكد من هذا، دعنا نأخذ المثال التالي:

```
1  var a;
2  if (a === undefined){
3       console.log('a is undefined');
4  }
5  else{
6      console.log('a is defined');
7  }
```

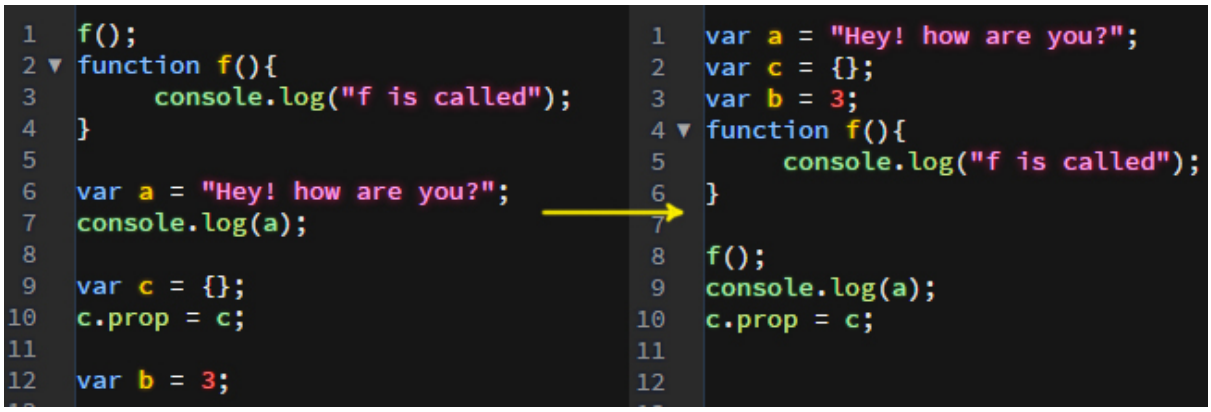
ستكون النتيجة بالطبع: a is undefined. و هذا لأننا أبلغنا عن المتغير a باستخدام الكلمة المفتاحية var دون نعطي له قيمة. في هذه الحالة سيتولى محرك جافا سكريبت الباقي، حيث أنه سيخصص مساحة لهذا المتغير، و يعطيه القيمة undefined مبدئيا. لذا عند عملية التحقق باستخدام if، فإن الشرط الأول صحيح، لأن قيمة a في هذه الحالة هي undefined.

إفرض أننا أعطينا قيمة للمتغير a أثناء الإبلاغ عنه، أي قبل عملية التحقق، مثلا:

var a = “something”

في هذه الحالة تستبدل القيمة undefined بالقيمة التي قدمناها، و هي في مثالنا هذا السلسلة النصية: ‘something’، إذن ستتحقق if من الشرط، و تجده خاطئا، و بهذا ستنتقل إلى else، ليكون الناتج: a is defined.

هناك أمر سائد يتم تداوله بين أغلب المبرمجين، و هو أن محرك جافا سكريبت ينقل المتغيرات، و الدوال إلى الأعلى قبل التنفيذ! أي هكذا:



و هذا مغاير تماما لما يحدث فعليا.

في الحقيقة يستعمل هذا المفهوم الخاطئ في أغلب المواقع، العربية منها و الغربية، لتوضيح سلوك جافا سكريبت خلف الكواليس، مما ولّد بعض الغموض لدى غالبية المبرمجين.

مرحلة التنفيذ:

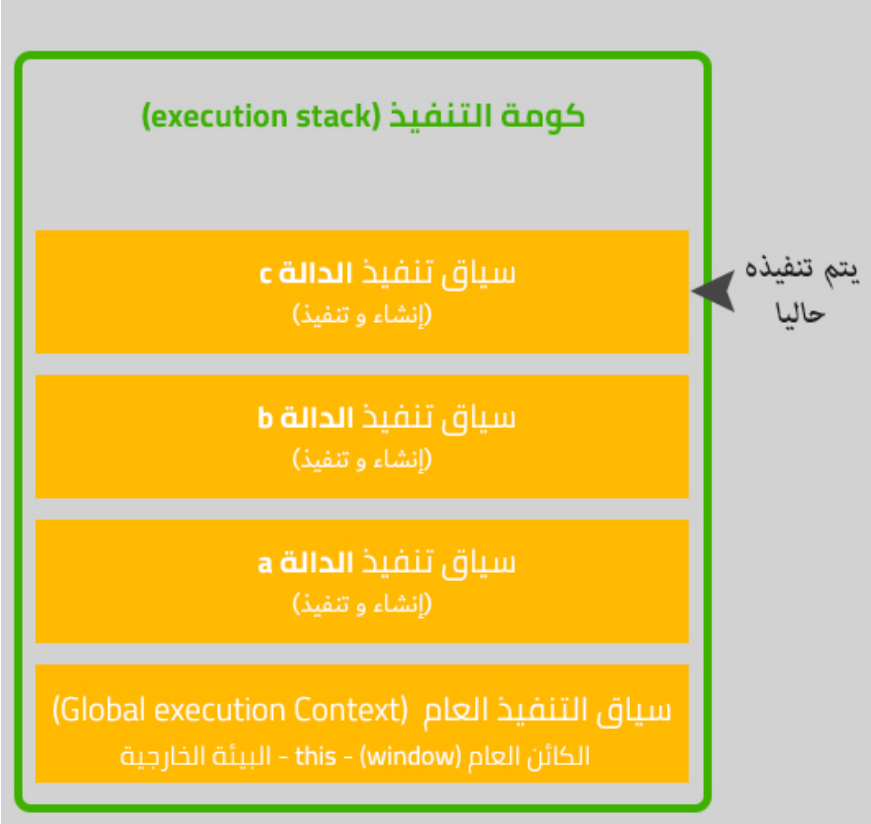
في هذه المرحلة يكون كل من الكائن العام و المتغير و البيئة الخارجية متوفرة. و الآن سيبدأ بتنفيذ الشيفرة التي كتبتها سطرا بسطر. حيث يقزم بتفسيرها، تحويلها، يجميعها و تنفيذها بطريقة يفهمها الحاسوب.

البيئة الخارجية:

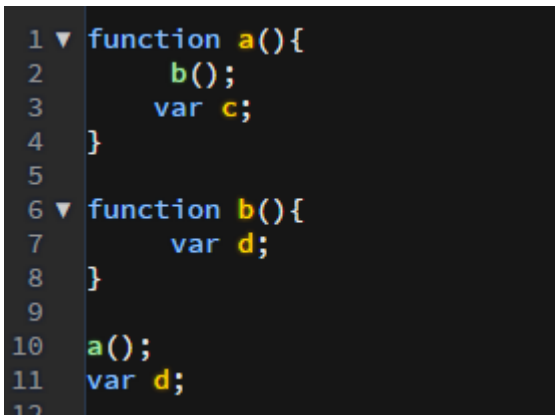
عند تنفيذ شيفرة موجودة بداخل دالة ما، فإن البيئة الخارجية، هي ما يوجد خارج الدالة؛ أما إذا تم تنفيذ الشيفرة في المستوى العام، أو البيئة العامة، ففي هذه الحالة لا توجد بيئة خارجية.

كومة التنفيذ:

لقد قلنا سابقا بأن المحلل اللغوي في مرحلة الإنشاء، يقوم بتحليل البرنامج، و عندما يأتي إلى الدوال، فإنه يضعها بكاملها في الذاكرة. حسنا، دعني أخبرك شيئا جديدا: في حالة إستدعاء دالة، فإن محرك جافا سكريبت يقوم بإنشاء سياق تنفيذ جديد لها، و يضعه في "كومة التنفيذ"، حيث تتكوّن كومة التنفيذ من مجموعة سياقات، بعضها فوق بعض. يتم تكديس السياق الجديد في الأعلى فوق سياق التنفيذ السابق، و السياق الذي يكون في الأعلى هو الذي يتم تنفيذه حاليا. إذن متى ما تم إستدعاء دالة ما، فإن سياق تنفيذ جديد يتم إنشائه، و يوضع في كومة التنفيذ فوق السياق السابق. شاهد هذا الرسم التوضيحي لتفهم:



في الواقع، لكل سياق مساحته الخاصة في الذاكرة للمتغيرات، و الدوال. و بعد تنفيذ جميع الأسطر في سياق التنفيذ الحالي سطرا بسطر كما قلنا سابقا، يتم إزالة هذا السياق من كومة التنفيذ؛ إذ سينتقل إلى السياق الذي تحته أي السياق السابق، و ينفذ الأسطر التي بداخله، و هكذا حتى يعود إلى السياق العام. و بالمثال يتضح المقال:



```
function a(){
    b();
    var c;
}

function b(){
    var d;
}

a();
var d;
```

لدينا في هذا المثال دالتين a، b و متغير d، إذن ما الذي سيحدث عند تشغيل هذا البرنامج؟

أولا، و كما نعرف سيُنشِأ محرك جافا سكريبت سياق التنفيذ العام، و بالطبع سينشأ لنا بشكل تلقائي الكائن العام window (ما دمنا نشغل البرنامج في المتصفح)، و المتغير this الذي يشير إلى الكائن العام window.

ثانيا، يقوم المحلل اللغوي بتحليل الشيفرة ليتحقق من صحة القواعد. و بما أنه قد مر على الدالتين a و b، فإنه سيتم تخزينهما بكاملهما في الذاكرة، و سيخصص المحرك مساحة للمتغير d في الذاكرة، و يعطيه القيمة undefined. إنتهت هنا مرحلة الإنشاء.

تأتي الآن مرحلة التنفيذ. بالطبع سيقوم المحرك بمحاولة تنفيذ البرنامج سطرا بسطر، إذن سيمر على تعريف الدالتين، و لكنه لن ينفذ أيا منهما، و لا ما بداخلهما، حتى يعثر على عملية الإستدعاء. إذن عندما يصل إلى السطر العاشر في مثالنا هذا، سيجد أننا إستدعينا الدالة a و هنا يحدث ما يلي:

سياق جديد خاص بهذه الدالة سيتم إنشائه، و يتم وضعه فيما يسمى "كومة التنفيذ"، فوق سياق التنفيذ العام. و كما قلنا سابقا أن السياق الموجود في الأعلى هو الذي سينفذ حاليا. و بما أن سياق الدالة a موجود في الأعلى إذن سيتم تنفيذه الآن. حسنا ما الذي تتوقع أن يحدث الآن؟

ستتم عمليتي الإنشاء، و التنفيذ في سياق التنفيذ الخاص بهذه الدالة، بالضبط كما حدث مع سياق التنفيذ العام، و سيمتلك مساحته الخاصة للمتغيرات و الدوال المعرفة داخل الدالة.

في مرحلة الإنشاء يشتغل المحلل اللغوي ليتحقق من صحة ما بداخل الدالة لغويا، و يبدأ المحرك بإعداد المساحة للدوال، و المتغيرات التي بداخلها، و طبعا سيعيّن القيمة undefined إلى كل متغيراتها مبدئيا.

حسنا، إنتهت مرحلة الإنشاء بالنسبة لسياق التنفيذ الخاص بالدالة a، يأتي الآن لمرحلة التنفيذ فيتصادف مع إستدعاء الدالة b، و من جديد يتم إنشاء سياق تنفيذ جديد خاص بالدالة b، و يضعه في الأعلى فوق سياق التنفيذ الخاص بالدالة a في كومة التنفيذ. و نفس الأمر يتم مع ما يخص الدالة b سيجد تعريف لمتغير إسمه d، يضعه في الذاكرة بالقيمة undefined و بعدها لا يجد ما ينفذه، إذن سيخرج من الدالة b و سيتم إزالة سياق تنفيذها من كومة.

و الآن، و بعد حذف سياق الدالة b يصبح سياق a هو السياق الأعلى، إذن سيتم تنفيذه في الوقت الحالي، و بما أنه نفّذ السطر الذي يستدعي الدالة b سينزل إلى السطر الموالي، و الذي يقوم بتعريف متغير c، يخصص له مساحة، و يضع له القيمة undefined. إنتهت الأسطر البرمجية في الدالة a و بهذا سيخرج منها، و ينزع أو يحذف سياقها من كومة التنفيذ. و ما تبقى الآن إلا السياق العام، و سيعود محرك جافا سكريبت إلى أين كان سابقا؛ في السطر الذي يستدعي الدالة a، هذا السطر قد تم الإنتهاء من تنفيذه، و بالطبع، سينتقل إلى السطر الموالي (السطر 11)، حيث سيعرّف المتغير d، يخصص له مساحة و يسند له القيمة undefined.

هكذا يتم تنفيذ البرنامج حسب هذا السيناريو الطويل في الشرح السريع في الإنجاز.

ملاحظة:

لا يهم ترتيب تعاريف الدوال، بالنسبة لسياقات التنفيذ الخاصة بها. المهم هو ترتيب إستدعائاتها.

لنفرض أن في المثال السابق كان *تعريف* الدالة b فوق *تعريف* الدالة a. دون أن نغيّر في ترتيب إستدعائيهما، فإن نفس العملية السابقة ستتم. سياق تنفيذ خاص ب الدالة a سينشأ، و يوضع في كومة التنفيذ فوق السياق العام. في مرحلة التنفيذ الخاصة بالدالة a سينشأ سياق التنفيذ الخاص بالدالة b و يوضع في الأعلى ليتم تنفيذ ما بداخلها، و يحذف ثم ينفذ ما تبقى من سياق التنفيذ الخاص بالدالة a، و يحذف إلى أن يصل إلى سياق التنفيذ العام.

خلاصة القول، أنه كل ما يتم إستدعاء دالة من أي مكان في البرنامج، حتى لو أن الدالة تستدعي نفسها بداخلها، فإن سياق تنفيذ جديد خاص بها سينشأ، و يوضع في الأعلى في كومة التنفيذ.

بيئة المتغير (Variable Environment):

عندما نتكلم عن بيئة المتغير، فنحن نتكلم عن مكان المتغيرات التي أنشأتها، و أين تعيش، و كيف ترتبط ببعضها البعض. فلنأخذ هذا المثال لنفهم:

```
1  function func2(){
2      var x;
3      console.log(x);
4  }
5
6  function func1(){
7      var x = "func1 var";
8      console.log(x);
9      func2();
10 }
11
12 var x = 4;
13 console.log(x);
14 func1();
15 console.log(x);
```

في هذا المثال، لدينا نفس الإسم لثلاث متغيرات معرفة في أماكن مختلفة، ماذا ستكون النتيجة عند التشغيل؟

سيتم طباعة الرقم 4، ثم العبارة "func1 var"، ثم undefined، و أخيرا الرقم 4 مرة أخرى.

و لكن لما لم تتغير قيمة المتغير x رغم أننا عرفناه عدة مرات و بقيم مختلفة!؟

حسنا هنا سيتضح معنى ***بيئة المتغير***.

دعنا نرى ماذا لدينا أولا:

لدينا دالتان func1، و func2

لدينا المتغير x في السطر 12، و الذي لا يتبع لأي دالة (معرّف في السياق العام).

إذن المتغير x في السطر 12 سيكون عام، أي أنه سيوضع سياق التنفيذ العام، و سيكون خاصية من خواص الكائن العام window.

عند تشغيل البرنامج، و أثناء مرحلة التنفيذ (بعد أن خصص محرك جافا سكريبت مساحة للمتغير x المعرّف في السطر 12 و أعطاه القيمة undefined مبدئيا)، يقوم المحرك بإعطاء، أو بالأحرى تغيير القيمة لهذا المتغير من undefined إلى القيمة 4، بعدها ينتقل إلى السطر الموالي ليطبع القيمة 4 في وحدة التحكم (console). ثم ينتقل إلى السطر الموالي.

في السطر (14) هناك إستدعاء للدالة func1، و كما عرفنا سابقا سيتم إنشاء سياق تنفيذ جديد بمساحته الخاصة للمتغيرات، و الدوال، و يوضع فوق السياق العام في كومة التنفيذ. و بالطبع

ستكون هناك مرحلتي الإنشاء، و التنفيذ. في مرحلة الإنشاء سيعثر على تعريف المتغير x و سيخصّص له مساحته الخاصة ضمن سياق التنفيذ لهذه الدالة. إذن فالمتغير x المعرّف في الدالة func1 ليس له علاقة بالمتغير x المعرّف خارجها (في السطر 12)، فذاك معرّف ضمن سياق الدالة، و هذا الأخير معرّف ضمن السياق العام.

في مرحلة التنفيذ سيخصص القيمة "func1 var" لهذا المتغير، ثم ينزل إلى السطر الموالي فيطبع قيمته إلى وحدة التحكم. ثم ينزل ليجد إستدعاءَ للدالة func2، الأمر واضح: إنشاء سياق تنفيذ خاص بهذه الدالة، وضعه فوق سياق الدالة السابقة، مرحلة إنشاء ثم مرحلة تنفيذ.

أثناء مرحلة الإنشاء يصادف تعريفا للمتغير x، يخصص له مساحة ضمن هذا السياق الجديد، وبما أننا لم نسند له أي قيمة، فإنه في مرحلة التنفيذ سيطبع القيمة undefined التي خُصصت له في مرحلة الإنشاء.

إنتهت الدالة func2، و بذلك سيحذف سياقها، ثم العودة إلى السياق السابق (لأنه السياق الأعلى في كومة التنفيذ) ليحذفه أيضا بعد أن إنتهى من تنفيذ ما بداخل الدالة func1، و في الأخير الرجوع إلى السياق العام لتنفيذ ما تبقى، إذن سينزل إلى السطر 14 ليطبع قيمة المتغير x. فأي قيمة ستُطبع؟ بالطبع إنها القيمة 4. لأنه حاليا في السياق العام، و سيبحث عن هذا المتغير في هذا السياق.

و لهذا فإن النتيجة ستكون:

```
4
func1 var
undefined
4
>
```

سلسلة النطاق (Scope Chain):

لنفهم سلسلة النطاق جيدا سنأخذ مثالا:

```
1  function func2(){
2      console.log(x);
3  }
4
5  function func1(){
6      var x = "func1 var";
7      func2();
8  }
9
10 var x = 4;
11 func1();
```

هذا المثال مشابه للسابق كثيرا إلا أننا حذفنا المتغير x من الدالة func2 في رأيك ما هي القيمة التي ستطبع في وحدة التحكم؟ هل هي القيمة undefined أم هي القيمة "func1 var" أم القيمة 4؟ دعنا نشاهد النتيجة:

```
4
>
```

ربما توقعت أن نحصل على خطأ لأننا لم نعرّف المتغير x داخل هذه الدالة، أو ربما يطبع undefined و ربما توقعت أن يطبع القيمة "func1 var" للمتغير x المعرف داخل func1 لأننا إستدعينا func2 من داخلها!

و لكن لماذا طبع 4؟ أوَلَمْ نقل من قبل أن لكل دالة سياق تنفيذ خاص بها حيث توضع المتغيرات والدوال المعرفة داخلها؟ إذن لماذا طبع قيمة المتغير المعرف في السطر العاشر؟

حسنا، ما قلناه سابقا صحيح، دعني أوضح لك مفهوما يتعلق بشيء ذكرناه سابقا، ألا و هو المرجعية أو الإشارة إلى البيئة الخارجية. حيث أن كل سياق تنفيذ له مرجع إلى بيئته الخارجية. و الآن ماهي البيئة الخارجية لكل سياق؟ مهلا لحظة! هل تتذكر البيئة المعجمية التي تحدثنا عنها سابقا حيث أن مكان كتابة الكود مهم؟

أنظر أين كتبنا الدالة func2، لقد كتبناها في البيئة العامة للبرنامج، فهي ليست معرفة داخلة دالة أخرى بالرغم من أننا إستدعيناها داخل func1، و لكنّنا هنا نهتم بمكان تعريفها بغض النظر عن الترتيب فلا يهم إن تم تعريفها في الأعلى أو الأسفل و إنما داخل ماذا؟

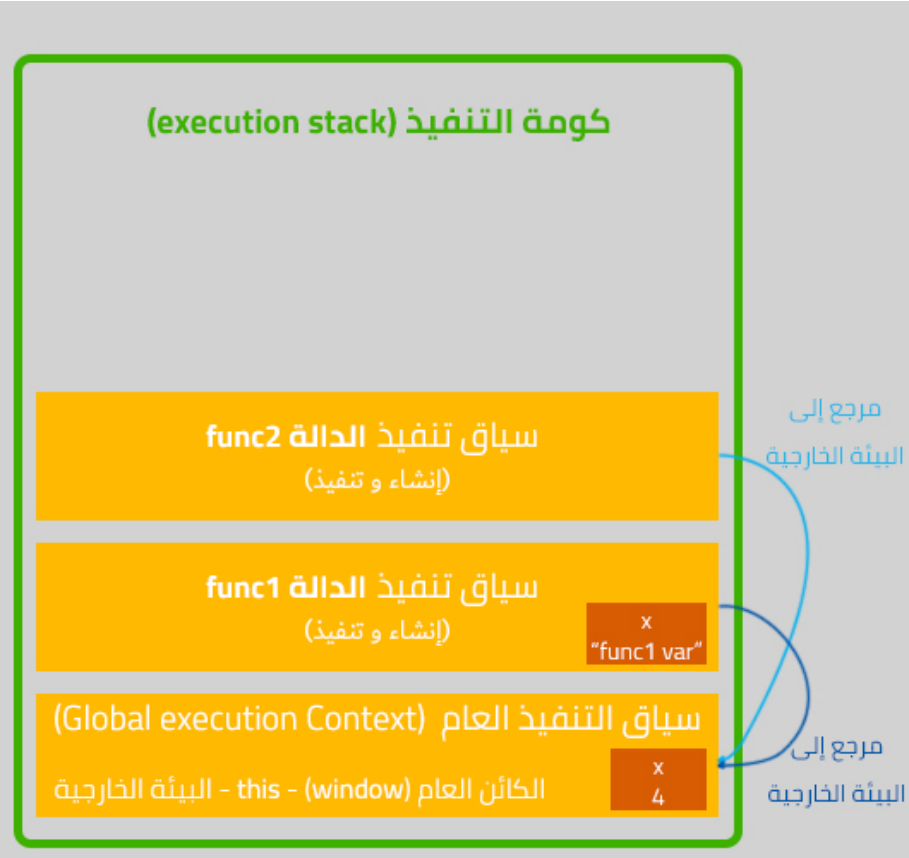
إهتمامنا بالمكان هنا لا يتعارض مع ما يحدث مع سياقات التنفيذ (حيث مكان التعريف غير مهم و ما يهم هو ترتيب الإستدعاء)!

تؤثر البنية المعجمية في إنشاء الإشارات إلى البيئات الخارجية لكل سياق تنفيذ. حيث تهتم جافا سكريبت بالبنية المعجمية لتربط سياق التنفيذ ببيئته الخارجية. فحينما تتعامل مع متغير داخل أي سياق تنفيذ معين، تبحث جافا سكريبت عن هذا المتغير في هذا السياق أولا، إذا لم تجده فإنها تنتقل إلى مرجع خارجي، و تبحث عنه هناك، حيث يكون هذا المرجع أسفل هذا السياق (الذي تم التعامل فيه مع المتغير) في كومة التنفيذ. و هذا المرجع يعتمد على بنية الدالة المعجمية أو بالأحرى مكان تعريفها؛

فبالنسبة للدالة func2 في مثالنا، و كذلك الدالة func1 فإن مرجعهما أو بيئتهما الخارجية هي سياق التنفيذ العام، و يعد هذا السياق أبعد بيئة خارجية يمكن الإشارة إليها، أو يمكن أن تكون مرجعا.

و لذلك عندما طلبنا طباعة قيمة المتغير x من داخل الدالة func2، و الذي لم نعرّفه بداخلها، فإن محرك جافا سكريبت بحث عنه في سياق التنفيذ الخاص بها أولا، فلم يجده، و أضطر إلى أن يبحث عنه في البيئة الخارجية الخاصة بها، التي هي سياق التنفيذ العام في هذه الحالة، و قد وجده معرّفا هناك بالفعل، و بهذا طبع لنا القيمة 4.

إليك الصورة التالية لتأخذ فكرة أوضح:

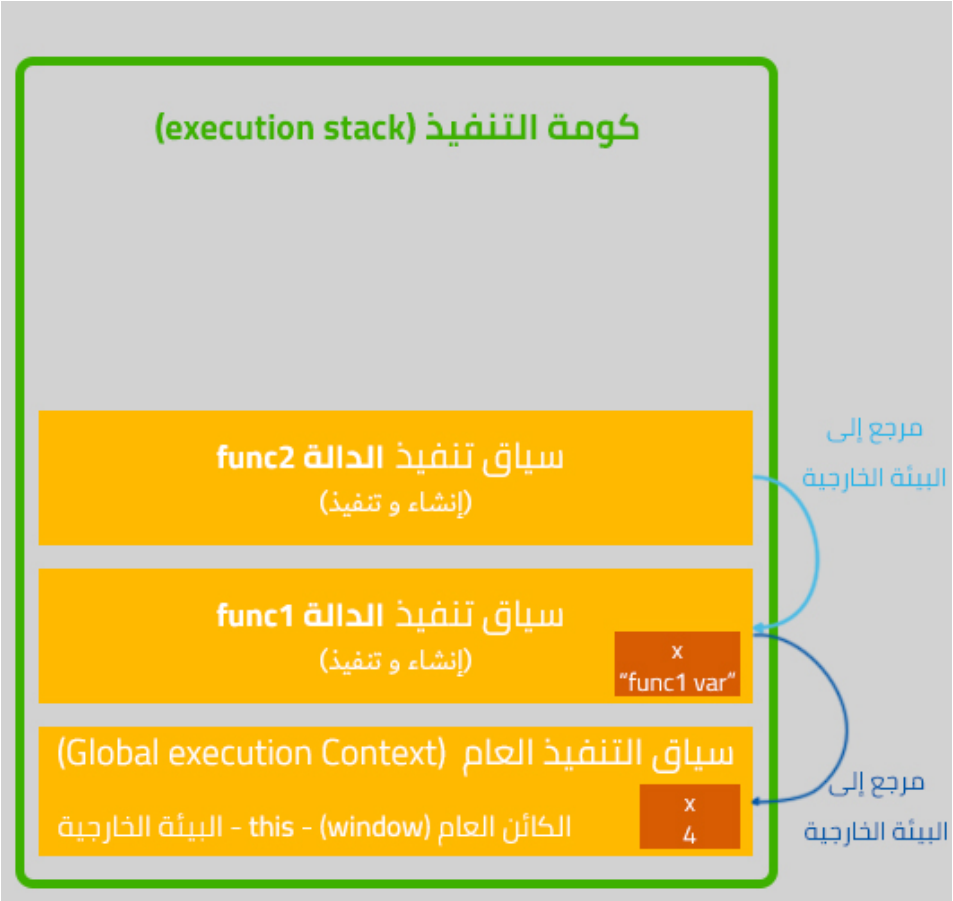


و الآن لنفرض أننا قمنا بتعريف الدالة func2 داخل الدالة func1، فماذا تتوقع أن تطبع لنا وحدة التحكم؟

أعتقد أنك ستقول القيمة "func1 var" وبالطبع هذا صحيح. لأن البنية المعجمية للدالة func2 قد تغيرت، و قد أصبحت جزءا من الدالة func1، و بالتالي فإن بيئتها الخارجية في هذه الحالة هي سياق تنفيذ الدالة func1:

```
function func1(){
    function func2(){
        console.log(x);
    }
    var x = "func1 var";
    func2();
}

var x = 4;
func1();
```

```
func1 var
>
```



لنغير المثال قليلا و لننظر على ماذا نحصل؟

```
function func1(){
    var x = "func1 var";
    func3();
    function func3(){
        var x = "func3 var";
        func4();
        function func4(){
            var x = 70;
            func2();
            function func2(){
                console.log(x);
            }
        }
    }
}

var x = 4;
func1();
```

```
70
>
```

لقد أصبح الأمر متداخلا قليلا، و لكن ماذا ستكون النتيجة يا ترى؟

حسنا ستكون 70 لأن البيئة الخارجية ل func2 هي سياق التنفيذ الخاص ب func4.

سنعدّل الكود قليلا مرة أخرى لنرى على ماذا نحصل:

```
function func1(){
    var x = "func1 var";
    func3();
    function func3(){
        var x = "func3 var";
        func4();
        function func4(){
            func2();
            function func2(){
                console.log(x);
            }
        }
    }
}

var x = 4;
func1();
```

قمنا بحذف المتغير المعرف في func4. و بهذا سيبحث محرك جافا سكريبت في سياق الدالة func2 عن المتغير x فلا يجده لينتقل إلى بيئتها الخارجية، و هي سياق الدالة func4 فلا يجده أيضا لينتقل إلى البيئة الخارجية الخاصة بسياق الدالة func4 ليجده هناك، فيرجع لنا قيمته مباشرة، و لا يهتم إلى بقية المتغيرات المشابهة. فأول ما يعثر على المتغير المطلوب يرجع قيمته. و هذا ما يسمى بسلسلة النطاق.

و أخيرا، أنظر المثال التالي:

```
function func1(){
    func3();
    function func3(){
        func4();
        function func4(){
            func2();
            function func2(){
                console.log(x);
            }
        }
    }
}
func1();
```

```
Uncaught ReferenceError: myVar is not defined
    at func2 (app.js:8)
    at func4 (app.js:6)
    at func3 (app.js:4)
    at func1 (app.js:2)
    at app.js:13
>
```

لاحظ أنه أعطانا خطأ مرجعية، و أخبرنا بأن x غير معرّف، و أنظر إلى التفاصيل تحت هذا الخطأ، حيث بحث في سياق func2، ثم func4، ثم func3، ثم func1، و أخيرا في السياق العام، فلم يجده في أي منها و لذلك أرجع لنا خطأ مرجعية.

و بهذا فإننا نستخلص أمرين و هما: أن البنية المعجمية مهمة في المرجعية؛ و ترتيب إستدعاء الدوال مهم في إنشاء سياقات التنفيذ الخاصة بها.

الكلمة المفتاحية let:

في الإصدار es6 من جافا سكريبت، تم إدخال كلمة مفتاحية جديدة تدعى let، حيث تعمل على تعريف المتغيرات مثلما تفعل var تماما، و يمكن أن تستعمل بدلها، و لكنّها لا تستبدلها البتة، فالكلمة var لا تزال باقية و تعمل، و هناك فرق بين هاتين الإثنتين، إذ أن الكلمة let تسمح لمحرك جافا سكريبت بإستعمال ما يسمى نطاق الكتلة، حيث يكون المتغير متوفرا ضمن الكتلة التي تم تعريفه بها، و من المعروف أن الكتلة هي ما بين القوسين المعقوفين {}، و ثانيا لن يكون متوفرا إلا بعد أن يتم تنفيذ السطر المعرّف به خلال مرحلة التنفيذ. لنأخذ مثالا ليسهل الفهم:

```
1 ▼ if (true){
2       console.log(c);
3       let c = "block scope";
4   }
5
```

```
Uncaught ReferenceError: c is not defined
    at app.js:2
>
```

أرأيت أنه أعطانا خطأً بدل أن يرجع القيمة undefined؟!

لكن لو جربنا أن نطبع القيمة بعد سطر التعريف سيرجع لنا القيمة بكل بساطة:

```
1 ▼ if (true){
2       let c = "block scope";
3       console.log(c);
4   }
5
```

```
block scope
>
```

حسنا، ماذا لو طبعنا قيمة c خارج كتلة العبارة if، ماذا ستكون النتيجة؟

```
1 ▼ if (true){
2       let c = "block scope";
3   }
4   console.log(c);
5
```

```
Uncaught ReferenceError: c is not defined
    at app.js:4
>
```

لقد حصلنا على خطأ، رغم أن المتغير لا يزال في الذاكرة، إلا أن المحرك غير مسموح له بإستعماله خارج حدود الكتلة المعرّف بها.

يجب أن تعلم أن الأمر يسري أيضا بالنسبة للحلقات for، و while، و أيّ كتلة في البرنامج.

النطاقات المزيفة:

التنفيذ المتزامن و اللامتزامن:

المتزامن هنا نقصد به: تنفيذ واحد في المرة الواحدة. أما **اللامتزامن** فيعني: عدة تنفيذات في المرة الواحدة. و بالنسبة إلى جافا سكريبت فإنها تزامنية.

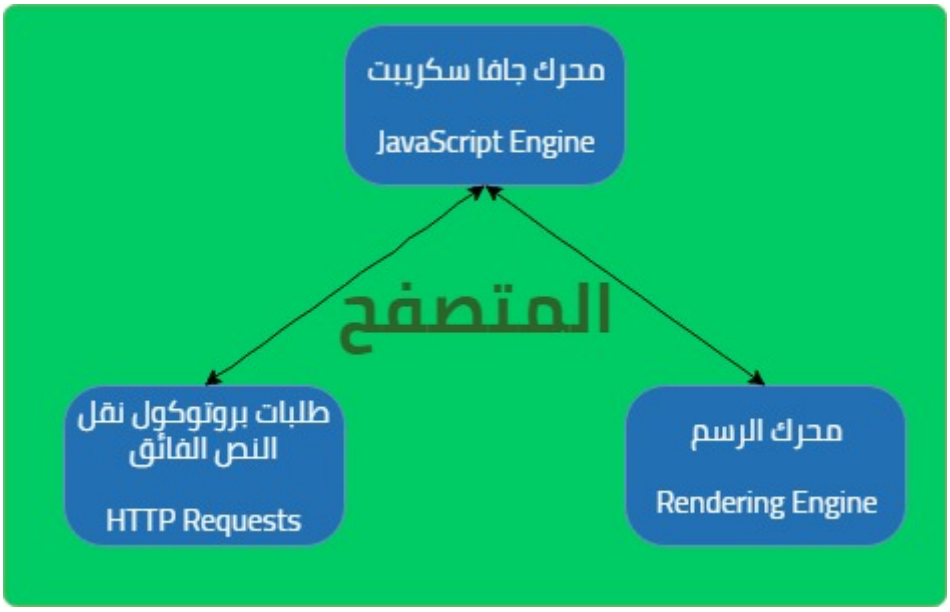
و لكن مهلا لحظة! ماذا فيما يخص الطلبات اللا متزامنة و التي تشتهر بها تقنية ajax!

إذا كيف نوفق بين هذين المفهومين؟ أو كيف تعالج جافا سكريبت هاته الأحداث الغير متزامنة أو تقنية ajax؟

حسنا، كما قلنا سابقا فإن جافا سكريبت تزامنية، أما فيما يخص التنفيذات اللامتزامنة فإن الأمر يتعلق بالمتصفح، أي خارج محرك جافا سكريبت، حيث أن محرك جافا سكريبت ليس هو الوحيد الموجود في مكونات المتصفح، فهناك محركات أخرى: كمحرك الرسم الذي يرسم صفحات الويب، و طلبات بروتوكول نقل النص الفائق (ب.ن.ن.ف - http)، متنصت الأحداث ... الخ.

و مع هذه العناصر الإضافية، يمتلك محرك جافا سكريبت روابط بينه و بينها، لكي يتمكن من التواصل معها فيتواصل مع محرك الرسم ليغير مظهر الصفحة مثلا أو يظهر تأثير الحركة... الخ، أو

يتواصل مع طلبات http ليجلب معلومات، أو يتواصل مع متنصت الأحداث ليعرف الحدث الذي تم فينفذ الدالة التي تخصه، و هكذا... دعنا نأخذ هذه الصورة للتوضيح:



و يحدث أن يعمل محرك جافا سكريبت، و المحركات الأخرى في نفس الوقت، أي أكثر من تنفيذ واحد يتم في الوقت الواحد، و هذا ما يدعى بالتنفيذ اللا متزامن، إلا أن هذا التنفيذ كما قلنا سابقا يحدث داخل **المتصفح**، و ما يُنفذ داخل محرك جافا سكريبت يتم تنفيذه تزامنيا، أي سطرا واحد في المرة الواحدة.

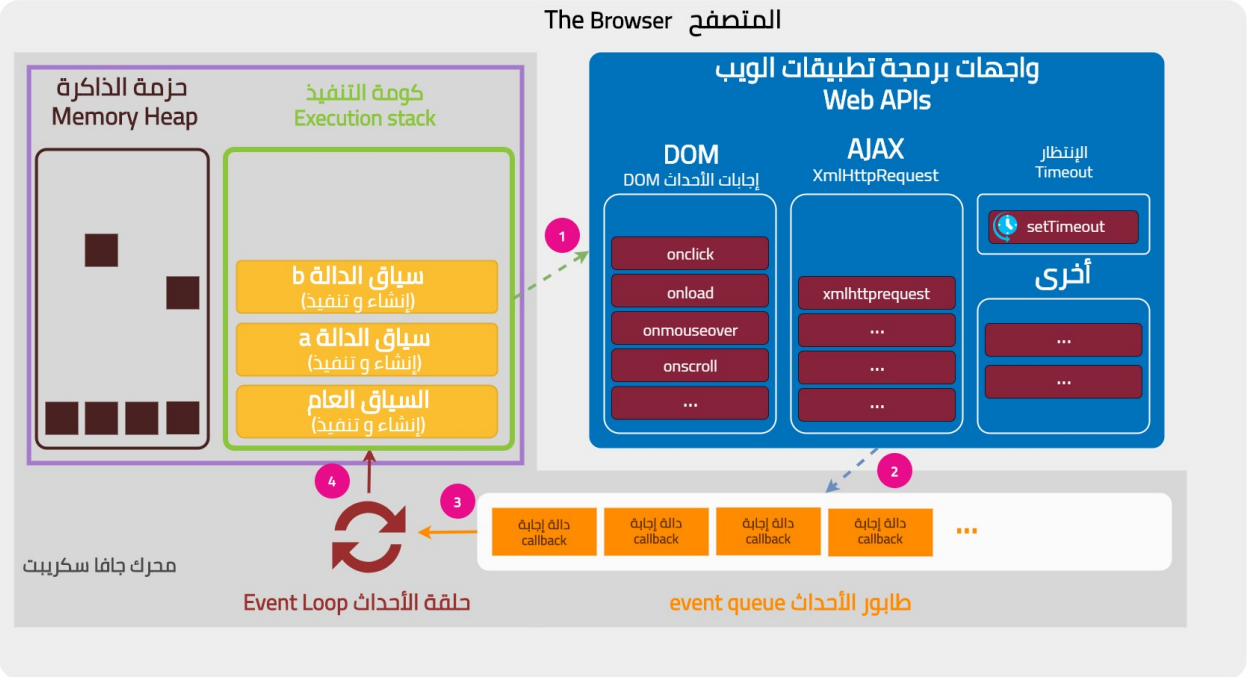
و لنقل مثلا أن دالة ستنفّذ عندما ينقر المستخدم زرا ما، و ما تقوم به هاته الدالة هو إرسال طلب (request) إلى الخادم بإستخدام تقنية ajax، حيث سيرسل هذا الطلب عبر محرك طلبات HTTP، في هذه الحالة فإن هذا الأخير و محرك جافا سكريبت يشتغلان في الوقت ذاته. هذا هو التنفيذ اللا متزامن. (*ما يحدث داخل المتصفح*).

تعلمنا سابقا كيف يتصرف محرك جافا سكريبت مع كومة التنفيذ، و كيف أن السياقات تُكدَّس بعضها فوق بعض، و السياق الذي إنتهى تنفيذ ما بداخله يتم إزالته. حسنا، هناك قائمة إضافية موجودة في محرك جافا سكريبت تدعى: طابور الأحداث.

ماذا يحدث عندما يطرأ حدث معين في الصفحة نريد أن نتنصت عليه كنقر زر مثلا، أو تمرير، أو إكتمال تحميل الصفحة... ؟

هناك في المتصفح ما يدعى بواجهات برمجة تطبيقات الويب (Web APIs)[1]، هاته الواجهات تعالج العديد من الأمور، منها الأحداث الحاصلة في المتصفح؛ حيث تترقب الصفحة لتقتنص الحدث و تنظر ما الدالة المرتبطة به فتضعها في طابور الأحداث، حيث تتالى الدوال داخل هذا الطابور واحدة تلو الأخرى في إنتظار تنفيذها.

بعد أن ينتهي محرك جافا سكريبت من تنفيذ ما بداخل كومة التنفيذ من شياقات و تفرغ تماما، حينها سيلقي نظرة على طابور الأحداث ليرى ما إذا كانت هناك دوال إجابة[2] (أو الرّد)، إن وجدت فإن الدالة الأولى تنتقل إلى كومة التنفيذ ليتم تنفيذها -فهذا هو دور كومة التنفيذ-، بعد الإنتهاء من تنفيذها تُرمى خارج الكومة و بالتالي تفرغ مجددا و تعاد الكرة مع الدالة التالية في الطابور؛ يدعى هذا العمل المتكرر بشكل دوري بحلقة الأحداث. إليك هذه الصورة للتوضيح:



لاحظ كيف تتم العملية بالترتيب، فما أن يُعثر على معالج (handler) خارج نطاق محرك جافا سكريبت فإنه يرسل إلى واجهة برمجة التطبيق (web API) الخاصة به كالـ AJAX مثلا أو ال DOM فيحلل هذا المعالج، و ينفذه، و يرجع النتيجة، و غالبا ما تكون دالة الإجابة التي حددناها سابقا لتُنفذ عند حدوث هذا الحدث، يضع المتصفح هاته الدالة في طابور الأحداث لتنتظر دورها في التنفيذ. فتنقلها حلقة الأحداث إلى كومة التنفيذ لتنفذ واحدة تلو الأخرى، و هكذا...

---

[1] واجهات برمجة التطبيق هي دوال جاهزة للإستخدام بدون الحاجة لمعرفة التفاصيل التي بداخلها، و أستعملت هذه الدوال من أجل إخفاء تلك التفاصيل و تسهيل القيام بمهام دون الحاجة إلى كتابتها مجددا عند الحاجة إليها.

[2] دالة الإجابة أو الرد (callback) ببساطة هي دالة ستنفذ بعدما ينتهي تنفيذ دالة أخرى، أما في الجافا سكريبت هي دوال يمكن تمريرها كوسيط و يمكن إرجاعها من دالة أخرى.

تذكر أنه لا تنفّذ الدوال التي في طابور الأحداث إلا بعدما تفرغ كومة التنفيذ تماما من تنفيذ كافة السياقات المكدّسة بداخلها. أرأيت أن محرك جافا سكريبت لا ينفّذ إلا عملية واحدة في المرة الواحدة! و لذلك قلنا عليه أنه متزامن.

هذا هو الفرق بين التنفيذ المتزامن، و الغير متزامن؛ فالأول خاص بمحرك جافا سكريبت، و الثاني خاص بالمتصفح.

لنأخذ مثال لمزيد من التوضيح:

```
// دالة تأجيل لخمس ثواني
function waiting(){
    var ms = 500 + new Date().getTime();
    while (new Date() < ms){}
    console.log("إنتهاء الدالة")
}

// معالج حدث النقر بالزر الأيسر
function clickHandler(){
    console.log('حدث نقر');
}

// التنصت على حدث النقر من أجل تنفيذ المعالج المخصص
document.addEventListener('click', clickHandler);

// إستدعاء دالة التأجيل
waiting();
// طباعة جملة تفيد بإنتهاء البرنامج أو إنتهاء السياق العام
console.log('إنتهاء التنفيذ');
```

في هذا المثال لدينا دالة للتأجيل 5 ثواني حيث تأخذ الوقت الحالي، و تضيف إليه 5 ثواني، و تخزّنه بمتغير، ثم تقارن الوقت الحالي مع المتغير عبر الحلقة while، و بهذا ستأخذ الصفحة خمس ثواني ليكتمل تحميلها.

بعدها عرفنا دالة كمعالج لحدث النقر، تطبع عبارة في وحدة التحكم.

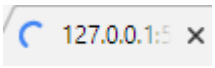
ربطنا الصفحة بمتنصت على حدث النقر لينفّذ لنا معالجنا السابق عند حدوث عملية نقر. و بعدها إستدعينا دالة التأجيل، و أخيرا نطبع عبارة لنعلم أن تنفيذ السياق العام قد إنتهى.

في رأيك ما هي النتيجة التي سنحصل عليها؟ و بأيّ ترتيب؟

دعنا نشغل البرنامج دون أن ننقر في الصفحة و نرى:

```
إنتهاء الدالة
إنتهاء التنفيذ
>
```

لقد تمت طباعة العبارتين بشكل عادي بعد أن علقّتْ الصفحة مدة خمس ثواني! لاحظ علامة التحميل:



لنعد تشغيل البرنامج مجددا، و هذه المرة ننقر في الصفحة خلال تلك الخمس ثواني. من ليس له إطلاع على ما يحدث في الخلفية سيتوقع أن تظهر عبارة دالة الإنتظار، ثم عبارة حدث النقر فالعبارة الدالة على إنتهاء البرنامج، و لكن في الواقع الأمر مخالف لذلك:

```
إنتهاء الدالة
إنتهاء التنفيذ
حدث نقر
>
```

أرأيت! لقد طبع عبارة حدث النقر في الأخير رغم أننا نقرنا في الصفحة خلال فترة التحميل. و هذا قد تم تنفيذه حسب ما شرحناه سابقا فيما يخص التنفيذ اللا متزامن.

جرّب الشيفرة بنفسك لتتأكد من صحة الكلام.

تلميح:

هناك موقع رائع وفّره المبرمج فيليب روبرتس Philip Roberts يوضح ما شرحناه سابقا بشكل مبطّأ لفهم الميكانيزم. هذا هو الرابط (في وقت كتابة هذا الكتاب):

http://latentflip.com/loupe/

أنواع البيانات:

لكل لغة برمجة طريقتها في التعامل مع البيانات المعروفة، كالسلاسل، و الأرقام ... الخ، و بالنظر إلى طريقة تعامل جافا سكريبت مع البيانات مقارنة ببعض اللغات الأخرى، فإن جافا سكريبت ديناميكية، و ليست ثابتة.

فأثناء تعريفك للمتغيرات فأنت لست بحاجة للإفصاح عن أنواعها، بل فقط تستعمل الكلمة المفتاحية var ثم إسم المتغير، و قيمته إن شئت، أو تتركه دون قيمة، هذا ما في الأمر.

و كما تعلم أن لدى جافا سكريبت ستة أنواع بدائية من المتغيرات، و نعني بالقيمة البدائية أنها تمثل قيمة واحدة فقط و بعبارة أخرى أنها لا تمثل كائنا، حيث أن الكائن هو مجموعة من القيم، و المفاتيح.

دعنا نعرج على هاته المتغيرات، و نشرحها:

1. **undefined:**
و تعني عدم وجود أي قيمة، و هي القيمة الإفتراضية التي يضعها محرك جافا سكريبت للمتغيرات كقيمة إبتدائية، حتى تخصَّص لها قيماً، و إلا فإنها ستبقى كقيمة لها.
و كتنبيه فإنك لست بحاجة لإعطائها كقيمة للمتغيرات أثناء تعريفك لها، بل دعها خاصة بالمحرك لتدلّ على أن المتغيّر لا يحمل أي قيمة بعد؛ يمكنك إجراء فحوصات لقيم المتغيرات ما إذا كانت تحمل هذه القيمة أم لا.

2. **null:**
لها نفس المعنى كسابقتها، إلا أنها أفضل للإستعمال من طرف المبرمج لدلالة على أن المتغير لا يحمل أي قيمة.

3. **boolean:**
نوع مشهور من البيانات بين مختلف لغات البرمجة، و هو إحدى القيمتين المنطقيتين إما صح أو خطأ true or false.

4. **number:**
في جافا سكريبت هناك نوع رقمي واحد للقيم يدعى: رقم، أو عدد، عكس لغات البرمجة الأخرى التي لديها عدة أنواع رقمية كالصحيح، و العشري، و العائم ...الخ. الرقم في جافا سكريبت عشري (أو ذو نقطة عائمة)، أي أنه دائما ملحق ببعض الكسور العشرية. و بالإضافة إلى أن الرقم يمثل عددا عشريا، فإن له ثلاث رموز أخرى و هي:
Infinity+، Infinity- و NaN (Not-a-Number).
يمكنك أن تتحقق في وحدة التحكم من نوع أحد هاته الرموز الثلاث، مثلا: typeof NaN.

5. **string:**
و هي سلسلة من المحارف محصورة بين علامتي إقتباس مفردة ' ' أو مزدوجة " ". و ليس بالضرورة أن تكون هاته المحارف مجرد حروف فقط، بل يمكن أن تكون خليطا من الحروف، و الأرقام و الرموز.

6. symbol:

نوع جديد من البيانات أدخل في الإصدار السادس من جافا سكريبت، أو إصدار ECMAScript 6 إختصارا ES6.

المعاملات:

المعاملات جزء لا يتجزأ من جل لغات البرمجة، و للمعاملات عدة أنواع، و أشكال، فمنها الحسابية، معاملات المقارنة، معاملات الإسناد، و المعاملات المنطقية. و كما نعرف فإن هذه المعاملات هي رموز تحتاج شيئين إثنين لإتمام العملية المطلوبة من حساب أو مقارنة... الخ. و لكن إليك هاته المعلومة الجديدة التي ستفاجئك!

المعاملات هي ***دوال خاصة*** مكتوبة "نحويا" بشكل مختلف عن الدوال العادية التي نكتبها في برامجنا. و بشكل عام فإن المعاملات تأخذ وسيطين (بارمترين)، و ترجع نتيجة واحدة؛ فلو نأخذ على سبيل المثال معامل الجمع +، حيث يأخذ رقمين كوسيطين و يرجع لنا حاصل الجمع:

```
var s = 9 + 1;
console.log(s);
```

```
10
>
```

الأمر سيان بالنسبة لبقية المعاملات، فمعاملات المقارنة ترجع لنا true أو false كنتيجة المقارنة بين الوسيطين المقدمين لها.

و الآن ماذا لو -فرضاً- أن معامل الجمع تم تعريف دالته كالتالي:

```
function +(a,b){
    return // حاصل الجمع
}
```

إذن لجمع رقمين سنستدعي معامل الجمع كالتالي:

```
+(9,1);
```

لكن كما ترى فإن هذا غير مألوف لنا عند جمع الأعداد، أو بالنسبة للعمليات الحسابية ككل، الأمر يبدو غريبا، و غير منطقي!

حسنا، في الآلات الحاسبة القديمة كانت تتم علمية الجمع، و الطرح المعروفة كالتالي: 8 5 + حيث نعطيها الرقم 8 ثم الرقم 5 ثم معامل الجمع في الأخير لتقوم الحاسبة بجمع الرقمين 8 و 5، و 7 ثم 2 ثم - لتقوم بطرح 2 من 7. تدعى هاته الطريقة بالترميز اللاحق (postfix notation)، حيث أن المعامل يلحق الرقمين أي يوضع في الأخير، و لو وضع في البداية مثلا: + 7 2 فإنها تدعى بالترميز البادئ (prefix notation)؛ أما الطريقة المعتادة التي نستعملها في حياتنا العامة، مثلا 7 + 2 فتدعى الترميز الوسطي (infix notation)، و هو الأكثر مناسبة لنا نحن البشر. و لهذا عمد مبرمجوا جافا سكريبت، و أغلب لغات البرمجة الأخرى على توفير هذا الترميز الأخير (الترميز الوسطي) الأكثر قابلية للقراءة بالنسبة للبشر، و السهل الفهم لتجنب أي تعقيدات نحن في غنى عنها.

و بالحديث عن المعاملات فإن هناك أمر ملازم لها لابد أن نتحدث عنه ألا و هو *أولوية المعاملات*.

أولوية المعاملات:

كما تعلمنا في الرياضيات أن علامتي الضرب، و القسمة لها أولوية على علامتي الجمع، و الطرح، كما أن للأقواس أولوية قصوى على الضرب، و القيمة، و هكذا ... الخ. فإن الأمر نفسه موجود في جافا سكريبت، و كافة لغات البرمجة؛ و السؤال الآن هو: كيف لجافا سكريبت أن تعرف أن هذا المعامل له أولوية على المعامل الآخر؟ فهي لا تملك عقلا لتقرر!

جواب هذا السؤال هو أن صنّاع جافا سكريبت حددوا الأولوية لكل معامل من خلال إعطائه رقما، و كلما كان الرقم كبيرا كانت الأولوية أعلى. فمثلا أولوية معامل الضرب هي 14 أما معامل الجمع فأولويته 13 و بما أن 14 أكبر من 13 فإن معامل الضرب له أولوية على معامل الجمع، و الأمر ينطبق على بقية المعاملات.

إذا أنت تلاحظ أن الأولوية تم تحديدها من قبل الصانع، أي أنه كان من الممكن أي يعطي للجمع أولوية على الضرب و بهذا ستختلف النتائج. لنأخذ العبارة التالية:

```
var a = 1 + 2 * 3
```

إن كان الضرب أولى من الجمع (و هي الحالة العادية) فالنتيجة هي: 7، حيث أولا سيتم إستدعاء دالة معامل الضرب مع تمرير الوسيطين 2 و 3 لترجع لنا هذه الدالة القيمة 6 فتصبح العبارة:

```
a = 1 + 6
```

و بعدها يتم إستدعاء دالة معامل الجمع مع تمرير الوسيطين 1 و 6 لترجع لنا هذه الدالة بدورها القيمة 7.

أما إن كان الجمع أولى من الضرب فالنتيجة هي: 9. حاول أن تكمل بقية السيناريو.

أرأيت أن النتائج تختلف بمجرد إختلاف الأولويات! الأمر كما في الحياة الواقعية أيضا.

و لكن أريد أن أنبهك لشيء آخر؛ لقد قلنا سابقا أن المعاملات هي دوال تختلف عن الدوال العادية في شكلها فقط. و لو نظرنا لهذه العبارة: var c = 3 + 5 + 7 فإننا نرى أن في السطر الواحد قد تم إستدعاء ثلاث دوال معاملات و هي دالة معامل الإسناد = و الذي يأخذ وسيطين بالطبع، ثم دالة معامل الجمع مرتين.
السؤال الذي يطرح نفسه الآن هو: كيف يتم تنفيذ هذا السطر؟ أو أي دوال المعاملات تستدعى أولا؟
صحيح أن في هذه الحالة النتيجة ستكون نفسها، إن جمعنا 3 و 5 ثم نجمع الناتج مع 7 أو نجمع 5 و 7 ثم نجمع الناتج مع 3. لكن ما يهمنا هنا هو ما الطريقة التي سيتّخذها محرك جافا سكريبت؟ و أي الدالتين ستُستدعى أولا، فلهما نفس الأولوية؟

للإجابة على هذا السؤال نحتاج إلى إضافة مفهوم آخر ألا و هو الترابطية (associativity)، (أو ما أحب أن أسميه "الإتجاهية" فقط للتسهيل) و التي تعني تحديد الطريقة التي يتم بها تحليل العوامل من نفس الأولوية، و لعلمك فإن هناك طريقتين من الترابطية و هما:
ترابطية يسارية أي (من اليسار إلى اليمين): و تعني أن أبعد حد من اليسار يتم إستدعاؤه أولا:
(a OP b) OP c OP d
ترابطية يمينية أي (من اليمين إلى اليسار): و تعني أن أبعد حد من اليمين يتم إستدعاؤه أولا:
a OP b OP (c OP d)

حيث: OP: معامل.

a, b, c, d: قيم.

لنعد إلى مثالنا السابق و نطبّق عليه هذا المفهوم:
الترابطية الخاصة بمعامل الجمع ذو الأولوية 13 (و كذلك معامل الطرح) هي ترابطية يسارية أي من اليسار إلى اليمين، أما معامل الإسناد فأولويته 3 و ترابطيته من اليمين إلى اليسار.
في هذه الحالة عندما يقرأ محرك جافا سكريبت هذا السطر و يأتي لتنفيذه، فإنه يرى أن أولوية معامل الجمع أعلى من أولوية الإسناد، لذلك دالة معامل الجمع ستستدعى أولا.
بعدها يجد معاملي جمع في هذا السطر، و هنا يلجأ إلى خاصية الترابطية ليكمل التنفيذ، و بما أن ترابطية معامل الجمع من اليسار إلى اليمين فإن المحرك سينفذ أقصى واحد على اليسار، أي أنه سينفذ 3 + 5 أولا، و عندما ترجع الدالة النتيجة، يهم بتنفيذ دالة معامل الجمع الثاني، فيمرّر النتيجة الأولى كوسيط مع الرقم 7 أي 8 + 7 لترجع الدالة النتيجة 15، و أخيرا يتم تنفيذ دالة

معامل الإسناد ذات الأولوية 3، و التي ترابطيتها من اليمين إلى اليسار، و بهذا تسند القيمة 15 التي في اليمين إلى المتغير a الذي في اليسار.

لنأخذ مثالا آخر لتعزيز الفهم:

```
var a = 3, b = 6, c = 2;
a = b = c;
console.log(a);
console.log(b);
console.log(c);
```

في هذا المثال ستكون قيم a، b و c متساوية، لكن ما هي تلك القيمة؟ هل هي 3 أم 6 أم 2؟ كما ترى، في السطر البرمجي الثاني يوجد معاملين من نفس الأولوية إذن سنستعين بخاصية الترابطية.
إن ترابطية معامل الإسناد من اليمين إلى اليسار، أي أن أقصى معامل في اليمين سينفذ أولا، و بهذا فإن b = c سيتم تنفيذها أولا، و طبعا هذا المعامل يسند ما في اليمين إلى ما في اليسار، إذن سيسند قيمة c إلى b أي b = 2. و بما أن دالة معامل الإسناد سترجع قيمة فإن القيمة المرجعة هي 2. أريدك أن تتحقق من هذا عبر وحدة التحكم و تطبع ما يلي: b = c (بعد تعريف المتغيرات طبعا!)
ستجد أن وحدة التحكم تطبع القيمة المُرجعة بعد هذه العبارة:

```
> b = c;
<- 2
```

إذن بعدما ينفَّذ أقصى معامل في اليمين يتم إرجاع قيمة، و هكذا سينتقل لإكمال التنفيذ حيث سيتبقى a = 2.

هل فهمت الطريقة؟ إنها بسيطة أليست كذلك؟ إلا أنها رغم بساطتها، يفتقدها أغلب المبرمجين مما يتركهم في حيرة من أمرهم حيال النتائج الغير متوقعة، و خصوصا مع تنقيح الأخطاء.

و للإلمام بالأولويات و الترابطيات لجميع المعاملات، هناك جدول وفرته لنا *شبكة مطور موزيلا* في إحدى صفحاتها:
developer mozilla network

و لتسهيل الأمر عليك، وفّرت لك ذلك الجدول مترجما للعربية في الصفحة التالية:
حيث الثلاث نقاط ... تعني شيفرة ما: رقم، أو سلسلة نصية، أو كائن ...الخ

الأولوية	نوع المعامل	الترابطية	المعامل
20	التجميع	-	(...)
19	الوصول إلى عضو	من اليسار إلى اليمين	... , ...
	الوصول إلى عضو معدود	من اليسار إلى اليمين	... [...]
	إنشاء كائن جديد (مع قائمة وسائط)	-	new ... (...)
	إستدعاء دالة	من اليسار إلى اليمين	... (...)
18	إنشاء كائن جديد (بدون قائمة وسائط)	من اليمين إلى اليسار	new ...
17	الزيادة اللاحقة	-	... ++
	النقصان اللاحقة		... --
16	لا المنطقية	من اليمين إلى اليسار	! ...
	لا البِتِّيَّة		~ ...
	زائد الأحادي		+ ...
	ناقص الأحادي		- ...
	الزيادة البادئة		++ ...
	الإنقاص البادئ		-- ...
	نوع كذا		typeof ...
	void		void ...
	الحذف		delete ...
	الإنتظار		await ...
15	الأس أو القوة	من اليمين إلى اليسار	... ** ...
14	الضرب	من اليسار إلى اليمين	... * ...
	القسمة		... / ...
	باقي القسمة		... % ...
13	الجمع	من اليسار إلى اليمين	... + ...
	الطرح		... - ...
12	تحويل البِتَّات يساراً	من اليسار إلى اليمين	... >> ...
	تحويل البِتَّات يميناً		... << ...
	تحويل البِتَّات يمينا الغير موقع		... <<< ...
11	أصغر من	من اليسار إلى اليمين	... > ...
	أصغر من أو يساوي		... => ...
	أكبر من		... < ...
	أكبر من أو يساوي		... =< ...
	في		... in ...
	حالة من - نسخة		... instanceof ...

10	المساواة	من اليسار إلى اليمين	... == ...
	عدم المساواة		... =! ...
	المساواة الصارمة		... === ...
	عدم المساواة الصارمة		... ==! ...
9	و البِتِّيَّة	من اليسار إلى اليمين	... & ...
8	لا أو البِتِّيَّة	من اليسار إلى اليمين	... ^ ...
7	أو البِتِّيَّة	من اليسار إلى اليمين	... / ...
6	و المنطقية	من اليسار إلى اليمين	... && ...
5	أو المنطقية	من اليسار إلى اليمين	... // ...
4	الشرط	من اليمين إلى اليسار	... ? ... : ...
3	الإسناد	من اليمين إلى اليسار	... = ...
			... += ...
			... -= ...
			... **= ...
			... *= ...
			... /= ...
			... %= ...
			... >>= ...
			... <<= ...
			... <<<= ...
			... &= ...
			... ^= ...
			... /= ...
2	الإخضاع	من اليمين إلى اليسار	yield ...
	الإخضاع*		yield* ...
1	الفاصلة \ التسلسل	من اليسار إلى اليمين	... , ...

حسنا، لنبقى مع معامل الجمع، و نرى ما الذي يقوم به عندما نمرر له وسيطين من نوع مختلف، مثلا:

```
var c = 4 + '4';
console.log(c);
```

قد يتوقع البعض أن تكون النتيجة 8، و لكن النتيجة التي سنحصل عليها قد تكون غريبة:

```
44
>
```

44؟ من أين أتى بهذه القيمة؟

في الواقع، النتيجة 44 ليست رقما، و إنما سلسلة نصية، و لكي تتأكد من ذلك تحقق من ذلك بإستخدام الدالة أو المعامل الأحادي typeof الذي يأخذ وسيطا واحدا و يرجع لنا سلسلة نصية توضح نوع الوسيط المُمرَّر لها، كالتالي:

```
> typeof c
<- "string"
> |
```

أرأيت ذلك؟ إذن كيف حصل هذا؟

كما نعلم أن جافا سكريبت ديناميكية، أي أنها لا تحتاج إلى تحديد نوع المتغير بل ستتعرف عليه وحدها، و بما أننا في هذا المثال مرّرنا لمعامل الجمع الرقم 4 كوسيط أول، و السلسلة النصية '4' كوسيط ثاني، هنا يأتي دور محرك جافا سكريبت ليقوم بتحويل الرقم 4 قسراً إلى سلسلة نصية '4'، و بعدها تتم عملية الجمع، و إرجاع النتيجة كالعادة، و هنا يجب أن ننوه إلى أن الرقم 4، و مثيله المحوّل قسرا (السلسلة النصية '4') ليسا الشيء نفسه في الذاكرة، فهما مختلفان تماما. كلٌّ في مكانه المخصص في الذاكرة.

و لو تلاحظ أننا لم نخبر جافا سكريبت بأن تحول الرقم إلى سلسلة نصية، لم نكتب أي شيئ من هذا القبيل. كل ما فعلناه هو ببساطة تمرير الرقم 4 و السلسلة النصية '4' كوسيطين لمعامل الجمع.

هذا لأن جافا سكريبت ديناميكية، و تميل إلى تحويل الرقم إلى سلسلة نصية بدل أن تعطينا خطأ، كما تفعل لغات البرمجة الأخرى، خصوصا من قدموا من لغة جافا، أو سي.

و فيما يخص خاصية التحويل القسري الذي تقوم به جافا سكريبت، دعنا نأخذ مثالا لتوضيح هذا الجانب أكثر:

لنأخذ هذا المثال:

```
1  console.log(1 < 2 < 3);
2
```

ما النتيجة المتوقعة من هذا المثال؟

لو حاولنا أن نتوقع النتيجة فربما سنقول أنها true أي صحيح أن 1 أصغر من 2 و 2 أصغر من 3:

```
true
>
```

و لكن ماذا لو عكسنا المتراجحة السابقة كالتالي:

```
console.log(3 < 2 < 1);
```

ماذا تتوقع في هذه الحالة؟ بالطبع ستقول `false` لأن 1 ليس أكبر من 2 و لا 2 أكبر 3. لذا دعنا نرى هل ستوافقنا جافا سكريبت الرأي أم لا:

```
true
>
```

true؟ هل هذا منطقي؟

من منطقنا البشري هذا خاطئ، أما من منطق جافا سكريبت فهذا صحيح! إذن كيف حدث هذا؟ لنستعمل ما تعلمناه سابقا فيما يخص أولوية المعامل، و ترابطيته:

لو لاحظنا في الجدول السابق بالنسبة لمعامل المقارنة < "أصغر من" فنجد أن أولويته 11 و لكنها لا تهمنا هنا لأن نفس المعامل في السطر الواحد، بل ما يهمنا هنا هو ترابطيته و التي هي يسارية أي من اليسار إلى اليمين؛ إذن 3 < 2 هي من ستعالج أولا لترجع لنا النتيجة false بالطبع، هنا يأتي الأمر الذي قد لا يدركه البعض، و هو خاصية التحويل القسري للقيم، التي تقوم بها جافا سكريبت، و لكي تفهم خاصية التحويل القسري هاته، لنجري بعض الفحوصات في وحدة التحكم:

من المعروف أن جافا سكريبت تمتلك دوالا مبنية بداخلها مسبقا منها الدوال Number()، Boolean()، String() ... إلخ. لنستعمل هاته الدوال، و نرى ماذا ستعطينا:

```
> Number(false)
< 0
> Number(true)
< 1
```

أنظر أننا لما مررنا القيمة false كوسيط للدالة Number أرجعت لنا القيمة 0، أما بالنسبة للقيمة true فالقيمة المرجعة هي 1. هذا الأمر يجعل معنى للمقارنات السابقة فبالنسبة إلى false < 1 فإن false تحول قسرا إلى القيمة 0 و بهذا تصبح المقارنة كالتالي 0 < 1 مما يجعلها صحيحة أي يتم إرجاع القيمة true. هذا الذي يحدث خلف الكواليس و يوّلد بعض الغموض مما يجعل جافا سكريبت تبدو غريبة و غير مفهومة!

ماذا لو مررنا سلسلة نصية تحتوي على ما يبدو أنه رقم للدالة Number()، أي كالتالي:

```
> Number("1");
< 1
> Number("-43");
< -43
```

أرأيت؟ لقد أرجعت لنا قيما رقمية بشكل طبيعي، حسنا دعنا نجرب شيئا آخر:

```
> Number("h");
< NaN
```

مهلا لحظة! ما الذي تعنيه القيمة NaN؟

NaN أو Not a Number و تعني "غير رقمي" أي: قيمة ليست رقمية.

و لو قارنّا هذه القيمة بالأرقام فالنتيجة دائما false:

```
> NaN < 1
false
> NaN > 1
false
> NaN == 0
false
```

أمثلة إضافية:

```
> Boolean(0);
false
> Boolean(1);
true
> Boolean(76);
true
> Boolean("");
false
> Boolean("d");
true
```

لاحظ إختلاف النتائج بإختلاف القيم، فالصفر و السلسلة النصية الفارغة تعطي false أما غير ذلك فإنه يعطي true.

```
> Number(undefined);
NaN
> Number(null);
0
```

لاحظ أن undefined لا يمكن تحويلها إلى قيمة رقمية، أما القيمة null فيقابلها الرقم 0. لذا قد يبدو لك في بادئ الأمر أن undefined و null هما الشيء ذاته إلا أنهما ليستا كذلك، فهما مختلفتان تماما.

هذه بعض الأمثلة للتوضيح فقط، و أترك لك حرية تجربة القيم السابقة مع الدالة Boolean() بنفسك لتكتشف المزيد. لا تتردد، فقط حاول!

ما أريد أن أنبهك إليه هو أن جافا سكريبت ديناميكية بطبعها فلا تحتاج إلى تحديد لنوع البيانات بل تهتم بهذا الجانب وحدها؛ إذ لديها خاصية التحويل القسري، فإذا ما تعارضت قيمتان في النوع فتحول إحداهما إلى نوع الآخر لتكمل عملية المقارنة، أو الإسناد بدل أن تزعجك برسالة خطأ. في الحقيقة هذه الميزة رائعة، و خطيرة في نفس الوقت، فمن روعتها ما ذكرناه آنفا، و من خطورتها هو إعطاء نتائج على نحو غير متوقع قد تصعّب من عملية التنقيح!

و لتجنب هذا المشكل فإنه ينصح بعدم مقارنة نوعين مختلفين من القيم إلا إذا أردت أنت ذلك بشكل صريح، و مقصود. و لا أنصحك أيضا بإستخدام الدوال السابقة لتحويل القيم قسرا من أجل مقارنتها!

هناك معاملين آخرين لديهما نفس السلوك و هما معامل المساواة المزدوج == و معامل عدم المساواة !=، فهما يستخدمان خاصية التحويل القسري في حالة إختلاف النوع مما قد يسبب نتائج غير متوقعة، و لتجنب هاته المشكلة يستحسن إستعمال عاملي المساوة و عدمها الصارمين أي === و !== فأول ما يجدان إختلافا في النوع يرجعان القيمة false، و لن يقوما بأي تحويل كان.

لهذا الأمر سأقترح عليك زيارة صفحة من *شبكة مطوري موزيلا* لمزيد من التفاصيل حول عمليات المقارنة، وقد إقتبست لك منها الجدول أدناه لتطّلع عليه، و تأخذ فكرة عن النتائج الممكنة:

رابط الصفحة:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

x	y	==	===	Object.is	SameValueZero
undefined	undefined	true	true	true	true
null	null	true	true	true	true
true	true	true	true	true	true
false	false	true	true	true	true
'foo'	'foo'	true	true	true	true
0	0	true	true	true	true
+0	-0	true	true	false	true
+0	0	true	true	true	true
-0	0	true	true	false	true
0	false	true	false	false	false
""	false	true	false	false	false
""	0	true	false	false	false
'0'	0	true	false	false	false
'17'	17	true	false	false	false
[1, 2]	'1,2'	true	false	false	false
new String('foo')	'foo'	true	false	false	false
null	undefined	true	false	false	false

x	y	==	===	Object.is	SameValueZero
null	false	false	false	false	false
undefined	false	false	false	false	false
{ foo: 'bar' }	{ foo: 'bar' }	false	false	false	false
new String('foo')	new String('foo')	false	false	false	false
0	null	false	false	false	false
0	NaN	false	false	false	false
'foo'	NaN	false	false	false	false
NaN	NaN	false	false	true	true

بخصوص Object.is فهذا أمر جديد في جافا سكريبت ES6 يمكنك الإطلاع عليه في شبكة مطوري موزيلا.

و الآن لنستعمل بعضا مما تعلمناه سابقا لنكتشف بعض الحيل التي يستعملها مبرمجو إطارات العمل، و المكاتب الشهيرة مثل jQuery و غيرها:

```
function greeting(name){
    console.log(name);
    console.log('hello ' + name);
}

greeting('Soufyane');
```

لدينا دالة بسيطة جدا، كل ما تقوم به هو طباعة الإسم و رسالة ترحيب.
لو نظرنا في وحدة التحكم سنجد النتيجة هناك:

```
Soufyane
hello Soufyane
>
```

حسنا، ماذا لو إستدعينا الدالة دون تمرير أي وسيط؟

```
function greeting(name){
    console.log(name);
    console.log('hello ' + name);
}

greeting();
```

```
undefined
hello undefined
>
```

لم نحصل على رسالة خطأ عكس لغات البرمجة الأخرى، بل تمت طباعة سلسلة -ليست جيدة في نظرنا- في وحدة التحكم.

لو تتذكر ما تحدثنا عنه بالنسبة إلى سياق التنفيذ فإن جافا سكريبت ستنشأ سياق الدالة greeting، و تنشأ المتغير name داخل هذا السياق، و تعطيه القيمة undefined كقيمة إبتدائية، و بما أننا لم نمرر قيمة أثناء إستدعاء الدالة فإن قيمة name لا تزال undefined و بالتالي لن نحصل على خطأ.

و أثناء تنفيذ سطر الطباعة إلى وحدة التحكم فإن ما بداخل القوسين يحتوي على عملية جمع بين قيمتين من نوع مختلف، و بما أن undefined ليست سلسلة نصية فإن عملية تحويل قسري ستتم لتحوّل undefined إلى 'undefined' و بهذا تُجمع مع السلسلة ' hello'، و تُرجع النتيجة لتطبع.

ليس أمر جيد أن يتم طباعة عبارة كهاته، إذن نحن بحاجة إلى تحديد قيمة إفتراضية في حال لم يتم توفير إسم. أعتقد أن إستعمال عبارة if هو أول ما يتبادر إلى أذهاننا، لكن ماذا لو أريتك حيلة بسيطة أفضل من عبارة if؟

أولا دعنا نصل إليها بالتدرّج. لو نعود إلى موضوع المعاملات، و بالتحديد المعاملات المنطقية؛ كل ما نعرفه عنها هو أنها تتعامل مع القيم المنطقية true و false، و تعيد لنا إحدى هاتين القيمتين كنتيجة:

```
> true || false
< true
> true && false
< false
```

و لكن هذين المعاملين يرجعان true أو false إذا كانت الوسائط true أو false. أما في حالة وسائط من نوع مختلف فإن النتيجة المرجعة تكون حسب ما يلي:

بالنسبة ل ||: ترجع -كنتيجة- أول وسيط يمكن أن يحوّل إلى true، و إلا ترجع الوسيط الآخر.

بالنسبة ل &&: ترجع -كنتيجة- أول وسيط يمكن أن يحوّل إلى false، و إلا ترجع الوسيط الآخر.

```
> 0 || 1
< 1
> "" || "hello"
< "hello"
> 1 && undefined
< undefined
```

كما تعلم فإن الوسائط تحول قسرا إلى ما يقابلها من قيم منطقية لكي تتم المقارنة.

حاول أن تجرب قيما مختلفة مع هذين المعاملين لتتحقق من طريقة عملهما.

إذن لنستغل هاته الميزة في شيفرتنا السابقة:

```
function greeting(name){
    name = name || 'Dear User!';
    console.log('hello ' + name);
}

greeting();

```

أظن أن الأمر واضح الآن، سطر واحد يوفّر علينا كتابة عدة أسطر فقط بإستغلال خاصية التحويل القسري، هذا ما كنت أتحدث عنه سابقا عن إستعمال هاته الميزة بحذر، و عن قصد.
و لعلمك يستعمل هذه الحيلة العديد من المبرمجين المحترفين.
لنأخذ مثالا آخر:

```
<html>
    <head>
        <title></title>
    </head>
    <body>
        <script src="JQuery.js" ></script>
        <script src="MyLibrary.js" ></script>
        <script src="app.js"></script>
    </body>
</html>
```

أضفنا في الملف الرئيسي مكتبتين MyLibrary1.js و MyLibrary2.js فوق ملف app.js.
هذا كل ما تحتويه المكتبتين:

MyLibrary2.js (learn) - Brackets

vigate Debug Help

```
var libraryName = "My second library";
```

MyLibrary1.js (learn) - Brackets

vigate Debug Help

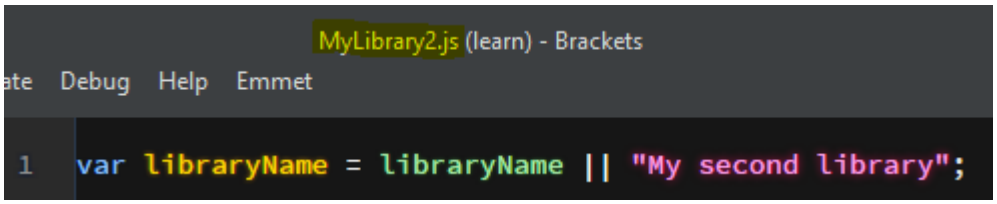
```
var libraryName = "My first library";
```

الفكرة هنا أننا أنشأنا مكتبتين و عرّفنا بداخلهما نفس المتغير لتخزين قيمة. ثم نحاول طباعة تلك القيمة من خلال ملف app.js. فأي القيمتين ستطبع؟ أحدهما أم كلاهما؟
يجب أن تعلم أن سطور ربط المكتبات بملف html لا تنشأ سياقا خاصا بكل ملف، و إنما تكدس الأكواد وراء بعضها البعض و كأنها كتبت في ملف واحد. لنرى النتيجة:

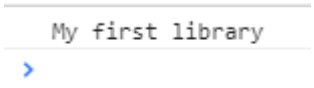
```
My second library
>
```

لقد طبع القيمة الخاصة بالمكتبة الثانية، و ذلك لأن الشيفرة الموجودة داخل هذه المكتبة وضعت بعد الشيفرة الموجودة في المكتبة الأولى، و بالتالي فإن قيمة المتغير في المكتبة الثانية تستبدل قيمة المتغير في المكتبة الأولى لأن libraryName معرّف في السياق العام حيث سيصبح خاصية من خواص الكائن العام window. و لهذا طبع لنا القيمة My second library.

هذه المشكلة قد تحدث أحيانا حينما ندرج مكتبات أخرى كتبها أشخاص آخرون عرّفوا بداخلها متغيرات و كائنات تؤدي أمرا ما و قد صادف أننا عرفنا متغيرات أو كائنات بنفس الإسم في مكتبتنا الخاصة التي أدرجناها بعد المكتبات السابقة، مما يؤدي إلى عدم إشتغال تلك المكتبات بسبب تغيّر قيم المتغيرات أو تركيبة الكائنات ... الخ. و لذا فإننا نلجأ إلى الحيلة السابقة لنتجنب حدوث أي تعارض، أو تغيير في القيم:



أرأيت! بسطر واحد حمينا أنفسنا من الوقوع في أخطاء قد يصعب تتبعها. و لو حدّثنا الصفحة سنرى ذلك:



قد ترى الكثير من السطور مثل السطر السابق و التي تستعمل لإعداد الكائن أو مجموعة الدوال التي تتكون منها المكتبة، أو إطار العمل، و لن يحدث أي شيئ إذا كان أحد الكائنات، أو الدوال، أو المتغيرات موجود بالفعل، و هذا لكي لا تحدث أية مشاكل.

الكائنات:

بعد حديثنا سابقا عن أنواع البيانات في جافا سكريبت، و التي تنقسم إلى نوعين: بدائية، و كائنية. حان الآن الوقت لنشرح بعض الجوانب حول الكائنات، و التي بفهمها ستتمكن من كسر الغموض المحيط بجافا سكريبت حول تصرفاتها الغريبة عند التعامل مع الكائنات.

في جافا سكريبت، كل شيئ عدا القيم البدائية عبارة عن كائن. هل هذا يعني أن المصفوفات، و الدوال كائنات أيضاً؟ نعم بالفعل هي كائنات، و سنرى هذا في الفصول القادمة.

من المعروف أنه لتعريف الكائن بحد ذاته في جافا سكريبت فإن هناك عدة طرق منها:

أولا: الطريقة الحرفية {}

ثانيا: إستخدام المعامل new مع الدوال.

ثالثا: object.create().

رغم أن الطرق الثلاث تؤدي نفس النتيجة، إلا أنه يفضل إستخدام الطريقة الأولى، و هي الطريقة الحرفية {} نظرا لسهولتها و قصرها مقارنة بالأُخريين، و شيوع إستعمالها بين المبرمجين أيضا.

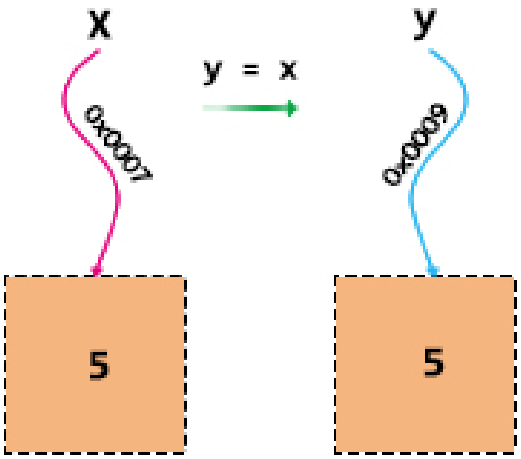
لنلقي نظرة على كيفية تعامل محرك جافا سكريبت مع الكائنات!

و قبل هذا لنرى كيف هو الحال مع القيم البدائية:

```
var x = 5;
var y;

y = x;
console.log('x = ' + x);
console.log('y = ' + y);

x = true;

console.log('x = ' + x);
console.log('y = ' + y);

```

```
x = 5
y = 5
x = true
y = 5
>
```

عرّفنا متغيرين x ذي القيمة البدائية 5، و y، بعدها حددنا قيمة y من خلال قيمة x. عند طباعة القيم في وحدة التحكم كانت النتيجة كما في الصورة أعلاه. بعد ذلك غيرنا قيمة x إلى قيمة أخرى true. ثم طبعنا قيمتي المتغيرين، و نلاحظ أنّ قيمة x تغيّرت أما قيمة y بقيت كما هي مما يعني أن قيمة y تم نسخها من قيمة x السابقة. و من هذا نعرف أن المتغيرات البدائية (التي تحمل قيم بدائية، و ليست كائنات) تتعامل بالقيم و نسخها.



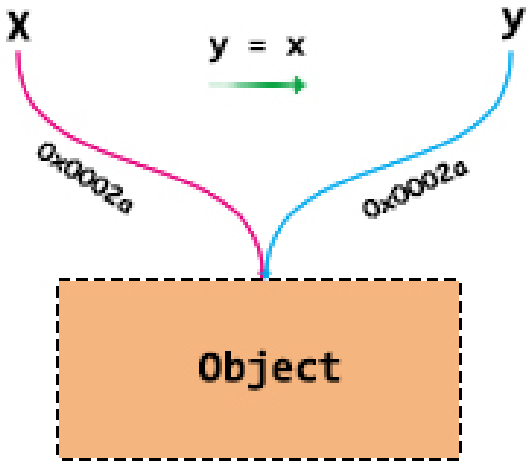
و الآن لنكرر الأمر نفسه مع الكائنات:

```
var x = { name: 'Soufyane'};
var y;

y = x;
console.log(x);
console.log(y);

x.name = 'Omar';

console.log(x);
console.log(y);

```

```
▶ {name: "Soufyane"}
▶ {name: "Soufyane"}
▶ {name: "Omar"}
▶ {name: "Omar"}
>
```

ما نلاحظه في هذا المثال أن قيمة y تغيّرت بتغيّر قيمة x!

إذن المتغيرات الكائنية أو لنقل الكائنات تتعامل بالمرجعية أو الإشارة حيث أن المتغيران x و y يشيران إلى نفس المكان الذي يعيش فيه الكائن في الذاكرة و الذي يعتبر كمرجع. فأي تغيير يطرأ على متغير فإن النتيجة تظهر على البقية. جرب مثلا أن تضيف خاصية جديدة إلى المتغير y و ستلاحظ أنها تظهر في المتغير x أيضا، و هذا لأن الكائن المرجع قد تغير، المرجعية تبدو كالتالي:



الأمر سيان بالنسبة للدوال، و المصفوفات أيضا، فهي كما قلنا تعتبر كائنات. جرب بنفسك لتحقق من الأمر.

و لعلمك فالتغيير يطرأ على الكائن من أي مكان يتم فيه ذلك حتى من داخل الدوال، أو كائنات أخرى، شاهد هذه الإضافة إلى المثال السابق:

```
12
13 function mutating(obj){
14     return obj.name = 'anonymous';
15 }
16
17 mutating(y);
18
19 console.log(x);
20 console.log(y);
21
```

```
{name: "anonymous"}
{name: "anonymous"}
>
```

```
22 var w = { c: true, d: y}
23 w.d.name = false;
24 console.log(x);
```

```
{name: false}
>
```

هذا جانب من أحد الجوانب التي تسبب بعض الغموض فيما يخص الكائنات. حيث أحيانا قد ترى نتائج مختلفة عما كنت تتوقع، و قد يكون عدم إستخدام المرجعية بالشكل الصحيح هو السبب! لذا فلتنتبه لهذه النقطة.

كل متغير يملك عنوان إلى القيمة المسندة له، فمثلا بالنسبة للمتغير x في المثال الأول فإنه يملك عنوان القيمة 5 و قد يكون ذاك العنوان على الشكل التالي: 0x0007 (هذا العنوان من أجل التوضيح فقط)

إضافة للمثال السابق:

```
x = {name: 'Mohammed'};

console.log(x);
console.log(y);
```

```
▶ {name: "Mohammed"}
▶ {name: false}
>
```

لاحظ هنا أننا قمنا بإعادة تعريف x، و في هذه الحالة x و y يشيران إلى كائنين مختلفين، رغم أن لهما خصائص متشابهة، حيث لكل منهما الخاصية name إلا أن كل منهما كائن قائم بحد ذاته.

إذن كل من x و y يملك عنوان يشير إلى المكان الذي يعيش فيه الكائن المرتبط به في الذاكرة.

فمعامل الإسناد هنا قد أنشئ مساحة جديدة في الذاكرة للكائن الجديد.

الدوال:

الآن حان وقت تفصيل جانب الدوال. أولا فلتعلم أن الدوال في جافا سكريبت هي دوال من الصنف الأول. و ماذا أقصد بالصنف الأول؟

الدوال من الصنف الأول تعني ببساطة تامة أن أي شيئ يمكنك فعله مع أنواع البيانات الأخرى يمكن أن فعله مع الدوال، حيث يمكن إسنادها إلى متغير، تمريرها كوسيط إلى دالة أخرى، إرجاعها من دالة، أو إنشاؤها على الطاير! (سنشرح هذه الجزئية لاحقا).

*جافا سكريبت ليس الوحيدة التي تملك الدوال من الصنف الأول، بل إشتهرت بها فقط.

لقد قلنا سابقا أن الدوال عبارة عن كائنات، و هذا معناه أنه يمكن أن تحتوي على خصائص، و دوال (طرق)! إلا أنها نوع خاص من الكائنات، لماذا؟ لأنها تمتلك جميع الميزات لكائن عادي بالإضافة إلى ميزات خاصة بها.

و من هذه الميزات أنها تملك بعض الخاصيات المخفية أهمها:

name:

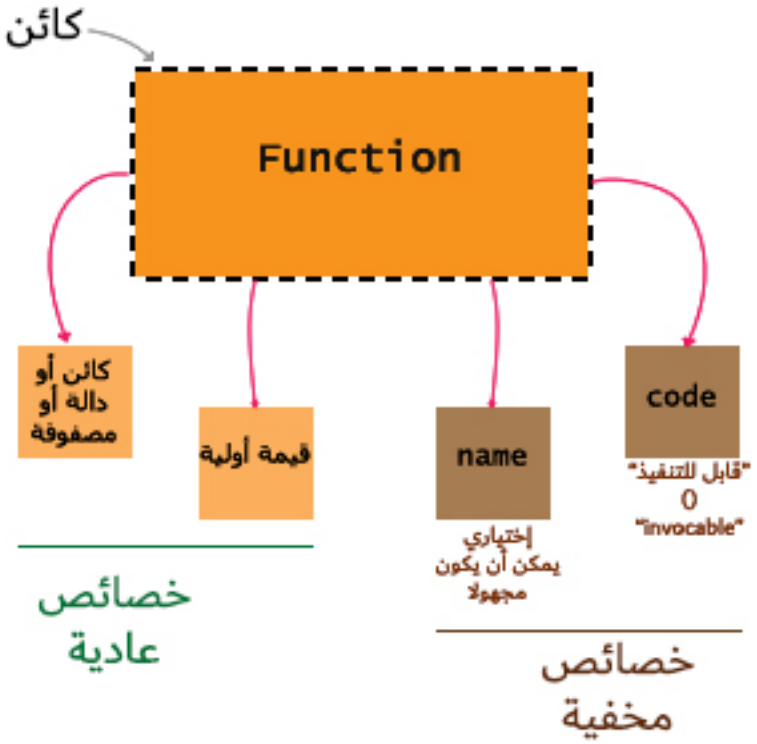
و هو أمر إختياري! نعم يمكن ألا تمتلك الدالة إسما و بهذا يطلق عليها دالة مجهولة. و يمكن أن تمتلك إسما و هذا أمر طبيعي.

code:

و هي السطور التي تكتبها داخل جسم الدالة، حيث أن هذه السطور أو الشيفرة توضع في خاصية لكائن الدالة و هي الخاصية code. هذه الخاصية لا يمكنك الوصول إليها.

إذن فالشيفرة التي تكتبها ليست هي الدالة بحد ذاتها، و إنما الدلة مجرد كائن يحمل خصائص و دوال (وظائف) و الشيفرة التي كتبتها داخلها هي واحدة من بين تلك الخصائص التي تضيفها لهذا الكائن.

و ما يميز هاته الخاصية أنها قابلة للتنفيذ.



فلنأخذ مثالا يوضح لنا ما سبق:

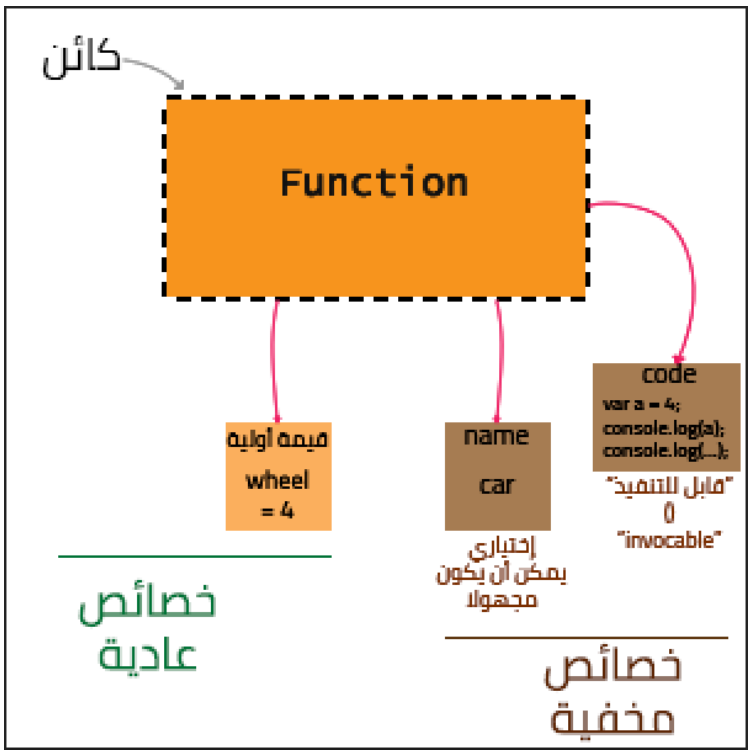
```
function car(){ // name هي قيمة الخاصية car
    //-----------------------------------------// <<<<<<<<<<<<<<
    var a = 5;                                  //    <== هذه الشيفرة   <<
    console.log(a);                             //     بداخل الخاصية    <<
    console.log('هذه الشيفرات موجودة داخل الخاصية code'); //    "code"    <<
    //-----------------------------------------// <<<<<<<<<<<<<<
}

car(); // "code" بإستدعاء الدالة يتم تنفيذ الخاصية

car.wheels = 4; // car إضافة خاصية إلى الكائن أو كائن الدالة

console.log(car.wheels);
```

```
5
هذه الشيفرات موجودة داخل الخاصية code
4
```

عرّفنا في هذا المثال كائن دالة حيث تحمل الخاصية name القيمة car، و الخاصية code تحمل الشيفرة الموجودة بداخل الدالة (التعليق يوضح ذلك). لذا عليك التوقف في إعتقاد أن الدالة عبارة عن حاوية لمجموعة من الشيفرات. بل هي مجرد كائن.

عند إستدعائنا للدالة بإستخدام القوسين، فإننا في الحقيقة ننفذ الخاصية code.

بعدها أضفنا الخاصية wheels كما نفعل مع أي كائن عادي.



و كما تعرف أن هناك بعض الوظائف، التي ربما إستخدمتها في برامجك أو قد رأيتها في مكان ما، أشهرها call، bind، apply و التي ربما تربك أغلبية المبرمجين الجدد أثناء التعامل معها.
سنعرج عليها لاحقا، و نشرحها بالتفصيل الممل كالعادة، فلا تقلق.

على كل حال، فإستخدام هاته الدوال يتم بطريقة التنقيط، مثلا myFunc.call(...)، myFunc.bind(...) و طريقة التنقيط هاته لا تتم إلا مع الكائنات، مما يوضح أن الدوال ماهي إلا كائنات. و هاته الطرق call و apply و bind هي وظائف معرّفة مسبقا في لغة جافا سكريبت. كما يمكنك أن تعرف طرقك الخاصة أيضا.

و الآن لنطلع على مفهومين يتعلقان بالدوال ألا و هما: تعبير الدالة، و تصريح أو جملة الدالة

تعبير الدالة (function expression):

التعبير بشكل عام هو أي وحدة من التعليمات البرمجية التي تُنتج، أو تنتهي بقيمة ما أثناء تقييمها، أو تنفيذها بغض النظر عن نوع هاته القيمة، مثلا العبارات التالية عبارة عن تعابير، فهي تنتهي بقيمة:

```
> 5 + 4;
< 9
> v = 'value';
< "value"
> o = {p: 'v'};
< ▶ {p: "v"}
```

حيث أن تقييم أو تنفيذ عملية الجمع إنتهى بالمجموع، و كذلك بالنسبة لعملية الإسناد إذ إنتهت بقيمة و هي السلسلة النصية 'value'، و قس على ذلك.
إذن تعبير الدالة هو الدالة التي تنتهي كقيمة لعملية ما:

```
> f = function(){
      console.log("I'm function Expretion");
  }
< ƒ (){
      console.log("I'm function Expretion");
  }
```

هنا عملية الإسناد إنتهت بالدالة المجهولة كقيمة.
أيضا يمكن القول بأن تعبير الدالة هي الدالة التي تكون جزء من تعبير.

تصريح الدالة (function statement):

فيما يخص التصريح أو الجملة بشكل عام، هي الشيفرة التي تقوم بعمل، و لا يمكن أن تنتهي كقيمة لأي عملية، مثلا: لا يمكن إسنادها إلى متغير.
فمثلا التصاريح أو الجمل الشرطية: if، for، while لا يمكن إسنادها إلى متغير، و لا تنتهي كقيمة لعملية ما، بل هي فقط تقوم بعمل، و هو التحقق من صحة الشرط، و تنفيذ الشيفرة الموافقة له. هذا كل ما في الأمر.

أما بالنسبة للدوال فإنه يمكننا أن نسندها إلى المتغيرات؟ هنا مربط الفرس! إذا ما أُسندت الدالة إلى متغير فإنها في هذه الحالة تدعى تعبير دالة، أما إن تم تعريفها لوحدها (التصريح بها) فهنا تدعى تصريح دالة، أي كالتالي:

```
> function statement(){
      console.log("I'm function statement");
  }
< undefined
```

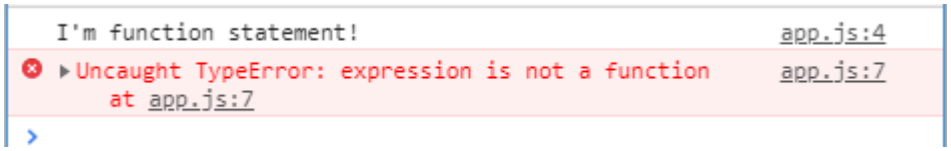
هنا الدالة تقوم بعمل.

و الآن أنظر لهذا الفرق حتى تفهم جيدا:

```
1   statement();
2
3 ▼ function statement(){
4       console.log("I'm function statement!");
5   }
6
7   expression();
8
9 ▼ var expression = function(){
10      console.log("I'm function expression!");
11  }
```

عرّفنا دالتين أولاهما تصريح، و الثانية تعبير. و قد إستدعينا كل منهما قبل تعريفها. إذا ما الذي سيحدث؟
أولا سنعود لفصل سياق التنفيذ، و لو تتذكر كيف يمر المحلل اللغوي عبر الشيفرة أثناء مرحلة الإنشاء، ما الذي يقوم به إذا تصادف مع الدوال؟ قلنا أنه يضعها كاملة في الذاكرة. و لكن في هذا المثال، هل سيضع كلتا الدالتين في الذاكرة؟
لو نظرنا في الشيفرة جيدا فإننا نرى أن الدالة الأولى عبارة عن تصريح، أي أن المحلل اللغوي بعد التحقق من صحة السطر الأول، يتصادف مع الكلمة المفتاحية function أولا فيعرف مباشرة أن هذه دالة إذ سيقوم بوضعها كاملة في الذاكرة بالطبع بعد أن يحلل كل سطورها للتأكد من صحة القواعد، و ما إلى ذلك. أما الثانية فهي مسندة إلى متغير، أي أن الأمر هنا إسناد قيمة إلى متغير من خلال دالة الإسناد =، و التي تحتاج في الأخير إلى تقييم أو تنفيذ حتى تتم عملية

الإسناد. معناه أن هذا الأمر سيتم في مرحلة التنفيذ، إذا في المرحلة الأولى مرحلة التحليل سيتم وضع القيمة المبدئية undefined إلى المتغير expression.
عند التنفيذ يجد في السطر الأول إستدعاءً للدالة statement و التي توجد في الذاكرة، و يملك عنوان الوصول إليها، فيتم تنفيذ هذا السطر بشكل عادي.
أما بالنسبة للسطر السابع إذ تتم عملية إستدعاء للدالة expression، و التي سيبحث عنها بذاك الإسم في الذاكرة فعلى ماذا سيعثر؟ سيعثر على المتغير expression ذي القيمة undefined التي سيعوضها مكان expression و يحاول تنفيذ السطر، فما الذي تتوقع أن يحدث إذا تمت معاملة القيمة undefined كدالة؟ بالطبع سنحصل على خطأ، و هذا ما سيخبرنا به محرك جافا سكريبت بأن expression ليس دالة!



لهذا لو أردنا أن نستدعي الدالة expression فإننا نستدعيها بعد سطر التعريف، حيث سيتمكن المحرك من الوصول إليها بعد تنفيذ سطر التعريف، و الإنتهاء بالدالة المجهولة كقيمة للمتغير expression.

في الحقيقة expression ليس إلا متغيرا يملك عنوان الدالة المجهولة على الذاكرة (حيث أنها لا تملك إسما (إسمها مجهول)، فهو يشير إليها فقط، و ليس دالة بحد ذاته! أما الدالة الأولى statement فيمكن الوصول إليها من خلال إسمها.

هذا هو الفرق بين *تصريح الدالة*، و *تعبير الدالة*.
قد يقول أحدهم أنه يمكن تسمية الدالة المسندة إلى المتغير expression. نعم هذا صحيح، و لكن يفضَّل عدم فعل ذلك.

إنشاء دالة على الطاير:

رأينا أنه لا يمكنك تعريف دالة مجهولة بتصريح دالة، فهذا ينتج خطأً، بسبب المحلل الذي ما إن يقرأ الحرف الأول f فيتوقع إما o أو u فإذا وجد u توقع بقية الحروف n، o، i، t، c، n و هكذا ... الخ، أما بالنسبة لتعبير الدالة فذلك ممكن، لكونها جزءاً من تعبير ما، كإسنادها إلى متغير مثلا، لذلك يكون التعريف صحيحا؛ إذن ألا يوحي لك هذا بفكرة؟ أنه لتعريف دالة مجهولة يجب ألا يبتدأ

السطر بالكلمة المفتاحية function و إلا إعتبرها تصريحا، إذن ماذا لو سبقنا الدالة بأمر آخر غير المتغير فنحن لا نريد أن نربطها بأي شيئ في هذه الحالة المهم أنها جزء من تعبير؟!
حسننا، هل تتذكر موضوع المعاملات؟ لحسن الحظ هناك معامل يساعدنا على فعل ما نريده، ألا و هو معامل التجميع (). لنلقي عليه نظرة تفصيلية:

معامل التجميع ():

يسمح هذا المعامل بإعطاء أولوية "للتعبير" المحدد بداخله على بقية المعاملات الأخرى؛ حيث يتم تجميع التعبيرات المراد تنفيذها أولا في التعبير الرئيسي بين قوسين ()، و لأن له الأولية القصوى من بين كل المعاملات فإن أول ما ينفذ في أي تعبير هو ما بداخل هذا المعامل طبعا بعد إتباع نوع الترابطية إذا وجد نفس المعامل مرات عديدة، و كما أشرنا سابقا يعتبر ما بداخله وسيطا لأن هذا المعامل دالة.

```
var a = (4 + 2) * (2 - 1) * 3;
console.log(a);
```

```
18
```

هناك أشياء أخرى حول هذا المعامل:

* يمكنك إستعمال معاملات تجميع متداخلة في بعضها البعض كالتالي:

```
> ((( 3 + 7 )))
< 10
```

لاحظ أننا مررنا التعبير 3 + 7 كوسيط لدالة معامل التجميع، أو لنقل ببساطة قمنا بتجميع تلك العبارة، و التي بدورها قمنا بتجميعها مرة ثانية، و ثالثة، و حتى عاشرة لو أردنا، و كانت النتيجة مساوية لنتيجة التعبير السابق دون أقواس أي 3 + 7 فقط.
و لو أردت أن تفهم كيف تم تنفيذ السطر السابق فإن محرك جافا سكريبت يعثر على معامل التجميع فينفذ ما بداخله ((3 + 7))، فيجد معامل تجميع ثاني فينفذ ما بداخله (3 + 7)، و هذا الأخير معامل تجميع أيضا لينفذ ما بداخله، و هو التعبير 3 + 7 الذي يرجع النتيجة 10، و تصبح النتيجة كالتالي (((10))) و سيرجع معامل التجميع الداخلي القيمة 10 إلى معامل التجميع الثاني أي ((10))، و الذي سيرجعها أيضا إلى معامل التجميع الأول (10)، و الذي سيرجعها بدوره كنتيجة نهائية.

* هذا المعامل لا يستعمل مع الأرقام فقط بل يستعمل لتجميع التعابير، و أقصد بالتعبير هنا أي شيئ يرجع قيمة ما كالتعبير السابق الذي أرجع لنا القيمة 10، و من هذا المنطلق فإنه يستعمل لخداع المحلل اللغوي فيما يخص الدوال المجهولة، و هذا ما نهدف إليه في هذا الفصل.

كل ما سنقوم به هو تغليف الدالة بداخل قوسين، أي سنجعل المحلل يفهم أن الشيفرة عبارة عن تعبير، و ليس تصريح، و بهذا فإنه عند محاولته تحليل الشيفرة سيصادف قوس الفتح أولا) و هو أمر صحيح كما رأينا قبلا، فيتوقع بعده تعبيرا، لذا عندما يمر على الكلمة المفتاحية function سيتوقع إما وجود أو عدم وجود إسم، و سيواصل التحقق من بقية القواعد إلى أن يصل إلى قوس الإغلاق (. و بهذا نكون قد تحايلنا على المحلل فينفذ الكود بشكل سليم.

و الآن كيف ننفذ هاته الدالة؟

لنعد إلى الدوال المجهولة المسندة إلى متغير. كنا لو أردنا أن ننفذ تلك الدوال فإننا ببساطة نرفق إسم المتغير الذي يشير إليها بقوسي التنفيذ كما نفعل مع أي دالة:

```
var f = function(){
    console.log('hello');
}

f();
```

و يمكنك أن ترفق قوسي التنفيذ في سطر تعريف الدالة، ثم بعدها تستدعي المتغير وحده كالتالي:

```
var f = function(username){
    return 'welcome ' + username;
}();

console.log(f);
```

```
welcome undefined                                    app.js:5
>
```

لو أرفقنا قوسي التنفيذ مع المتغير f فإننا سنحصل على خطأ:

```
var f = function(username){
    return 'welcome ' + username;
}();

console.log(f());
```

```
▸Uncaught TypeError: f is not a function             app.js:5
    at app.js:5
>
```

حسنا لنفعل نفس الأمر مع دالتنا المجهولة إذ أننا سننفذها مباشرة داخل معامل التجميع:

```
(function(username){
    console.log('welcome ' + username);
}()
)
```

```
welcome undefined                                    app.js:2
>
```

أرأيت لقد تم تنفيذ الدلة المجهولة بشكل مباشر، و لو أردنا أن نمرر معاملات لهاته الدالة فببساطة سنمررها بين قوسي الإستدعاء كما نفعل مع كافة الدوال:

```
(function(username){
    console.log('welcome ' + username);
}('Soufyane')
)
```

```
welcome Soufyane                                     app.js:2
>
```

هناك طريقة أخرى لتنفيذ هذه الدالة:
بعدما غلفناها داخل القوسين فإن هذا المعامل سيرجعها إذ أننا سنتمكن من إستدعائها بعد عملية الإرجاع أي أننا سنضع قوسي الإستدعاء بعد معامل التجميع مباشرة:

```
(function(username){
    console.log('welcome ' + username);
}
)('Omar')
```

```
welcome Omar                                         app.js:2
>
```

يوجد إختلاف بين المبرمجين حول أي الطريقتين أمثل، و أكثر منطقية، مع أنه ليس هناك أية فروقات من حيث العمل، بل الإختلاف في الشكل، و التنسيق فقط، فبعضهم يقول أنه يجب أن تغلف كل شيئ داخل القوسين أي الدالة، و استدعائها، و البعض الآخر يقول أنه يجب تغليف الدالة داخل القوسين، و يتم الإستدعاء خارجا ليبدو الأمر أكثر منطقية!
في الحقيقة كِلا الطريقتين صحيحتين، و تؤديان نفس النتيجة. فأختر أيهما تناسبك، و إستعملها في شيفرتك.

لعلمك قد تصادف مثل هاته الحلية كثيرا في أغلب المكتبات و أطر العمل، و من بينها أشهر المكتبات على الإطلاق مكتبة جي كويري jQuery، و هناك مكتبة underscore أيضا و حتى

angular ... الخ، لتتأكد قم بتنزيل إحدى هاته المكتبات و أطلع على شيفرتها المصدرية، لترى أن الشيفرة توضع داخل دالة مجهولة، و يتم تغليفها داخل قوسين، أي تستعمل معامل التجميع لخداع المحلل اللغوي.

و الآن لنعد خطوة إلى الوراء أين قلنا أن الدوال عبارة عن كائن يحمل تلك الخصائص و الطرق، لنلقي نظرة على ما يحدث عند تنفيذ دالة ما أي عندما تنفذ الخاصية code، حيث هذا يحدد كيف يتم تنفيذ الشيفرة التي بداخلها. فماذا يحدث عند تنفيذها؟ نحن نعلم أنه يتم إنشاء سياق جديد كل مرة تستدعى فيها الدالة، (يمكننا أن نعتبر أن سياق التنفيذ يركز على الخاصية code لكائن الدالة)، و نعلم أن كل سياق تنفيذ له بيئة المتغيرات أين تخزن المتغيرات المُنشأة داخل الدالة، و هناك البيئة الخارجية أين سيتم البحث عن المتغيرات التي لا توجد داخل جسم الدالة، هناك أمران إضافيان، و لنقل معاملان ضمنيان يضبطهما محرك جافا سكريبت لنا بشكل تلقائي دون أي تدخل منا، هما المتغير this و شبه المصفوفة arguments.

هذان الأمران مفيدان لنا بشكل كبير لمساعدتنا على القيام ببعض العمليات.

المتغير this:

ربما قد تعاملت مع المتغير this من قبل، بالنسبة لهذا المتغير فإنه متوفر في كل سياق تنفيذ، و دائما يشير إلى كائن محدد، أي أنه متوفر في سياق التنفيذ العام أيضا حيث يشير إلى الكائن العام، إذ في حالة المتصفحات، الكائن العام هو window، أما بالنسبة للبيئات الأخرى فالكائن العام يختلف من بيئة لأخرى.

```
> this
◂ ▸Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, …}
>
```

لو جربنا أن نرى إلى ما يشير المتغير this من داخل دالة فعلى ماذا سنحصل؟

```
function f(){
    console.log(this);
}

f();
```

```
▸Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, …}
>
```

نلاحظ أنه يشير إلى الكائن العام window أيضا.

هناك أمر آخر يعتمد عليه هذا المتغير داخل الدوال، و هو نوع الوضع المستعمل أهو الوضع العادي أم الوضع الصارم 'strict mode'، لنرى ذلك:

في حالة الوضع العادي كما في المثال السابق كان المتغير this يشير إلى الكائن العام window،

أما حين نجرب الوضع الصارم 'use strict' سنحصل على نتيجة مختلفة:

```
function f(){
    'use strict'
    console.log(this);
    }
f();
```

```
top  Filter  Default levels  G
undefined                    app.js:3
>
```

لاحظ أنه عند إستعمال الوضع الصارم أصبحت قيمة المتغير this هي القيمة undefined!

حسنا ماذا لو جربنا نفس الأمر مع وظيفة خاصة بكائن؟

```
var obj = {
    prop: 'obj property',
    method: function(){
        console.log(this);
    },
    strictMethod: function(){
        'use strict'
        console.log(this);
    }
}

obj.method();
obj.strictMethod();
```

```
                                        app.js:12
▶ {prop: "obj property", method: ƒ, strictMethod: ƒ}
                                        app.js:16
▶ {prop: "obj property", method: ƒ, strictMethod: ƒ}
>
```

نرى أن المتغير this في هذه الحالة يشير إلى الكائن obj نفسه بغض النظر عن نوع الوضع المستعمل! و بهذا نستنتج أن المتغير this يشير إلى الكائن العام window إذا أستدعي في السياق العام أو من داخل دالة سواءً كانت تصريحا أو تعبيرا إذا كان الوضع عاديا، و في حالة

الوضع الصارم فالقيمة undefined هي قيمة هذا المتغير. أما إذا أستدعي من داخل وظيفة فهو يشير إلى الكائن الذي يحتويها.

إذن لو إستدعينا الدالة f بطريقة التنقيط على الكائن العام و بإستعمال الوضع الصارم، فإلى ماذا سيشير المتغير this؟ فكما نعلم أن أي متغير، أو دالة تعرّف في السياق العام ستصبح خاصية تابعة للكائن العام، أي أن الدالة f في هذه الحالة يمكن إستدعائها كالتالي:

```
function f(){
    'use strict'
    console.log(this);
    }
f();

window.f();
```

```
top    Filter    Default levels    G
undefined                                app.js:3
                                         app.js:3
Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, fr
ames: Window, …}
```

شاهد هذا الإختلاف في النتائج رغم أن الدالة نفسها، إلا أن طريقة الإستدعاء تؤثر على تحديد ما يشير إليه المتغير this، فالإستدعاء الأول كان إستدعاء لدالة، أما الثاني فهو إستدعاء لوظيفة تابعة لكائن.

هنا بدأت تتضح الرؤية، لكن ماذا عن هذه الحالة:

```
var obj = {
    prop: 'obj property',
    method: function(){
        console.log(this);
        var fn =  function(){
            console.log(this);
        }
        fn();
    }
}

obj.method();
```

```
{prop: "obj property", method: ƒ}
Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, …}
```

نلاحظ أن المتغير this الخاص بالدالة fn المعرفة و المستدعاة داخل الطريقة method يشير إلى الكائن العام window!! لكن هذا يبدوا غريبا نوعا ما! فمن المفترض أن يشير إلى الكائن obj لأن الدالة معرفة داخل وظيفة خاصة به! فكيف حدث هذا؟

حسنا، لو دققنا قليلا فقط في الشيفرة و إسترجعنا ما قلناه سابقا حول المتغير this لتمكنا من معرفة سبب هذا السلوك الغريب، هذا المتغير يشير إلى الكائن التي أستدعيت عليه الدالة كما هو الحال مع method أما بالنسبة للدالة fn فهي صحيح معرفة داخل الكائن obj إلا أنها ليست وظيفة من وظائفه أي أنه لا يمكننا أن نستدعيها كالتالي:

```
> obj.fn()
Uncaught TypeError: obj.fn is not a function
    at <anonymous>:1:5
>
```

فكما ترى أننا حصلنا على خطأ.

إذن هل ستصبح تابعة للكائن window؟

```
> fn()
Uncaught ReferenceError: fn is not defined
    at <anonymous>:1:1
```

أيضا لا يمكننا إستدعائها، فهي ليست تابعة لهذا الكائن أيضا، إذن كيف أصبح المتغير this يشير إلى الكائن العام؟

إذن هنا في هذه الحالة fn دالة. و لنتأكد من ذلك لنجرب الوضع الصارم و نرى النتيجة:

```
var obj = {
    prop: 'obj property',
    method: function(){
        'use strict'
        console.log(this);
        var fn =  function(){
            console.log(this);
        }
        fn();
    }
}

obj.method();
```

```
▶ {prop: "obj property", method: ƒ}        app.js:13
undefined                                  app.js:15
>
```

هل رأيت هذا؟ بعد إستعمال الوضع الصارم أصبحت القيمة undefinded هي قيمة المتغير this مما يدل على أن fn دالة و ليست وظيفة.

لحد الآن فهمنا طريقة عمل المتغير this مع الدوال، في السياق العام و مع الوظائف. لكن أليس من الأفضل لو نستطيع تحديد ما يشير إليه هذا المتغير بدلا من أن يتولى المحرك ذلك؟ في الواقع، نعم، يمكننا ذلك بكل بساطة، و ذلك عبر إحدى ثلاث طرق تابعة لكائنات الدوال ألا و هي bind، apply و call، حيث ستسمح لنا كل واحدة من هذه الطرق بتخصيص المتغير this بطريقة معينة، لنرى كيف يتم ذلك:

لنبدأ بالدالة **bind**:

```
var target = {
    name: 'Target',
    fn: function(){
        console.log('target method');
    }
};

function myFunc(){
    console.log( this.name );
    console.log( this.fn() );
}
```

عرّفنا في هذا المثال كائن بالإسم target له الخاصية name و الطريقة fn، و بعدها عرّفنا الدالة myFunc التي تطبع قيما إلى وحدة التحكم حسب ما هو موضح في الصورة.
لو إستدعينا الدلة myFunc لوحدها فإننا سنحصل على خطأ و ذلك لأن المتغير this يشير إلى الكائن العام في هذه الحالة، و بالتالي سيبحث على الخاصية name و الطريقة fn فلا يجدهما، و سيعوض كل واحدة منهما بالقيمة undefined أما بالنسبة للخاصية name فسيضيفها للكائن العام، و بالنسبة للطريقة fn فسيحاول تنفيذها و undefined ليست دالة:

```
myFunc();
```

```
                                                    app.js:9
▶ Uncaught TypeError: this.fn is not a              app.js:10
function
    at myFunc (app.js:10)
    at app.js:13
>
```

```
▶ myFunc: ƒ myFunc()
  name: ""
▶ navigator: Navigator {ver
  onabort: null
```

إذن لنخصص المتغير this حتى يشير إلى الكائن الذي نريد، فما علينا إلا الإستعانة بالطريقة bind، و التي تتوفر لدى جميع كائنات الدوال، تقبل هاته الدالة وسيطا واحدا، و هو عبارة عن الكائن الذي سيشير إليه المتغير this.

```
12
13  var fnCopy = myFunc.bind(target);
14  fnCopy();
```

تقوم الطريقة bind بإرجاع نسخة من الدالة المستدعاة عليها الوظيفة bind (كائن الدالة myFunc في المثال)، أي أن الدالة الأصلية لن تتأثر، و سيبقى المتغير this يشير إلى الكائن العام في الحالة العادية، و بالتالي ستكون النتيجة كالتالي:

```
Target                    app.js:9
target method             app.js:4
undefined                 app.js:10
>
```

ماذا لو أردنا أن نجعل الدالة myFunc تشير دائما إلى الكائن target؟

كل ما علينا فعله هو إستدعاء الطريقة bind على الدالة myFunc أثناء تعريفها كتعبير لا تصريح، و بالتالي سنستدعي الدالة myFunc لوحدها كالتالي:

```
8  var myFunc = function(){
9      console.log( this.name );
10     console.log( this.fn() );
11 }.bind(target);
12
13 var fnCopy = myFunc();
14 fnCopy();
```

لماذا لم نستدعي الطريقة bind على تصريح الدالة myFunc؟ لأنه في تلك الحالة سنحصل على خطأ بسبب عدم توقع المحرك وجود نقطة بعد قوس الإغلاق المعقوف. جرب و شاهد النتيجة لتتحقق.

في حالة تتطلّب الدالة الأصلية وسائط، فإننا نمررها لنسختها المرجعة من الطريقة bind، أي سنمررها إلى الدالة fnCopy.

ماذا لو لأردنا أن ننفذ الدلة مباشرة، و لا نريد إرجاع أية نسخة؟ هنا يجب أن نستعين إما بالوظيفة call أو الطريقة apply.

الوظيفة **call**:

هاته الوظيفة مختلفة قليلا عن سابقتها حيث تقوم بتنفيذ الدالة مباشرة، تستقبل هاته الوظيفة أكثر من وسيط، الأول يجب أن يكون عبارة عن الكائن الذي سيشير إليه المتغير this، و بقية الوسائط هي الوسائط التي ستمرر إلى الدالة الأصيلة، شاهد المثال التالي:

```
var myFunc = function(p1, p2){
    console.log(p1);
    console.log(p2);
    console.log( this.fn() );
};

myFunc.call(target, 'hello', 23);
```

```
hello                                   app.js:9
23                                      app.js:10
target method                           app.js:4
undefined                               app.js:11
>
```

الوظيفة **apply**:

تتصرف هاته الوظيفة مثل أختها call تماما، إلا أن الفرق بينهما هو أن تمرير الوسائط ل apply يكون عبر مصفوفة:

```
var myFunc = function(p1, p2){
    console.log(p1);
    console.log(p2);
    console.log( this.fn() );
};

myFunc.apply(target, ['hello', 23]);
```

```
hello                                   app.js:9
23                                      app.js:10
target method                           app.js:4
undefined                               app.js:11
>
```

لا فرق بين الوظيفتين call و apply إلّا في تمرير الوسائط، فقط يبقى لك الخيار في الأسلوب الذي تحب إستعماله في شيفرتك.

الشبه مصفوفة arguments:

و الآن نأتي لشرح المتغير الضمني الثاني بالنسبة للدوال، و هو arguments، فائدة هذا المتغير هو تخزين الوسائط الممرّرة للدالة ليسهل لنا إجراء عمليات عليها، كعدّها، أو تغييرها، و ما إلى ذلك:

```
function f(a, b, c){
    console.log(arguments);
}

f('hello', 12, true);
```

```
app.js:2
▼Arguments(3) ["hello", 12, true, callee: ƒ, Symbol(Symbol.iterator): ƒ]
   0: "hello"
   1: 12
   2: true
  ▶callee: ƒ f(a, b, c)
   length: 3
  ▶Symbol(Symbol.iterator): ƒ values()
  ▶__proto__: Object
```

هذا المتغير ليس مصفوفة، و إنما شبه مصفوفة نظرا لشبهه بالمصفوفات في بعض الخصائص، و ليس كلها. حيث له خاصية الطول مثلا لعد العناصر (أنظر المثال أعلاه)، فلو أردنا أن نعرف عدد الوسائط الممررة فإننا نستعين بهذا المعامل:

```
function f(a, b, c){
    console.log(arguments.length);
}

f('assalam', 'marhaba','hello', 'hi', 'hola');
```

```
5                                    app.js:2
>
```

و يمكن الوصول إلى العناصر عبر معامل الحاضنات [ ] كما نفعل مع المصفوفات:

```
function f(a, b, c){
    console.log(arguments[1]);
}

f('assalam', 'marhaba','hello', 'hi', 'hola');
```

```
marhaba                              app.js:2
>
```

قد يفيدنا هذا المتغير في حالة لم تمرر أية وسائط للدالة فنجعل الدلة تتوقف دون إطلاق خطأ، أو تقوم بعمل آخر:

```
function f(a, b, c){
    if (arguments.length === 0){
        console.log('there is no parameters!!!');
        return;
    }
    else{
        console.log(arguments[2]);
    }
}

f();
```

```
there is no parameters!!!                    app.js:3
>
```

ستجد هذا المتغير مستعمل بكثرة في أغلب المكتبات، و أطر العمل. و لكن في النسخة القادمة من جافا سكريبت ES6 سيتم إهماله مع أنه سيبقى متوفرا، و لكن ليس من المستحسن إستخدامه؛ إذ سيتم إدخال واحد جديد يدعى spread و الذي سيسمح بتمرير وسائط إضافية دون التصريح بها خلال تعريف الدالة و يتم ذلك بتعريف وسيط يبتدأ بثلاث نقط، كالتالي:

```
function f(a, ...args){
    console.log(a);
    console.log(args);
    console.log(args[0]);
}

f('assalam', 'marhaba','hello', 'hi', 'hola');
```

```
assalam                                      app.js:2
▶ (4) ["marhaba", "hello", "hi", "hola"]     app.js:3
marhaba                                      app.js:4
>
```

كما ترى تم تعريف وسيطين في تصريح الدالة الأول وسيط عادي، و الثاني هو الوسيط ...args و الذي سيخزن بقية الوسائط الممررة أثناء إستدعاء الدالة و ذلك على شكل مصفوفة. و للوصول إلى تلك الوسائط، فما عليك إلا إستعمال إسم الوسيط دون النقاط الثلاث، كما ترى في المثال أعلاه. (يمكنك إختيار الإسم الذي يناسبك لهذا الوسيط المهم أن تسبقه بثلاث نقاط).

و سيتم أيضا توفير ميزة القيمة الإفتراضية:

```
function f(a, b, c = 'anonymous'){
    console.log(a);
    console.log(b);
    console.log(c);
}

f();
```

هذه الميزة لا تدعمها المتصفحات القديمة، و ستتوفر في الجديدة إذ ستسهل عملية تحديد القيمة الإفتراضية بدل الإستعانة بالمعامل "أو" || كما فعلنا سابقا، إلا أنه الحل المناسب للمتصفحات القديمة التي لا تدعم آخر إصدارات جافا سكريبت الجديدة.
هناك ميزة أخرى ستضاف،

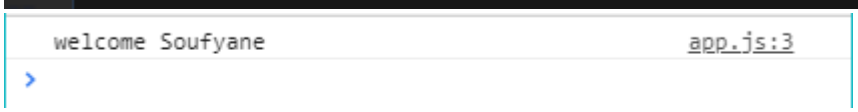
مرة أخرى هذه الميزات ستتوفر في النسخة القادمة من جافا سكريبت ES6 و التي ستدعمها المتصفحات الجديدة في المستقبل. ربما قد تدعمها بعض المتصفحات الأخيرة حاليا كمتصفح كروم الإصدار 69.0.3497.100 و الذي أستعمل في فحص و عرض نتائج الأمثلة في هذا الكتاب. لذا قد لا يدعمها المتصفح الذي تستعمله خصوصا إذا كان قديما بعض الشيء.

و الآن لننتقل إلى موضوع آخر يخص الدوال، وجب توضيحه، ألا و هو موضوع الدوال المنغلقة closures.

الدوال المنغلقة (closures):

موضوع الدوال المنغلقة موضوع سيئ السمعة بين المبرمجين لما به من غموض، و لكنه في الحقيقة غاية في البساطة، فقط إفهم ما يحدث خلف الكواليس، و بإذن الله ستتمكن من الموضوع، و بالمثال يتضح المقال:

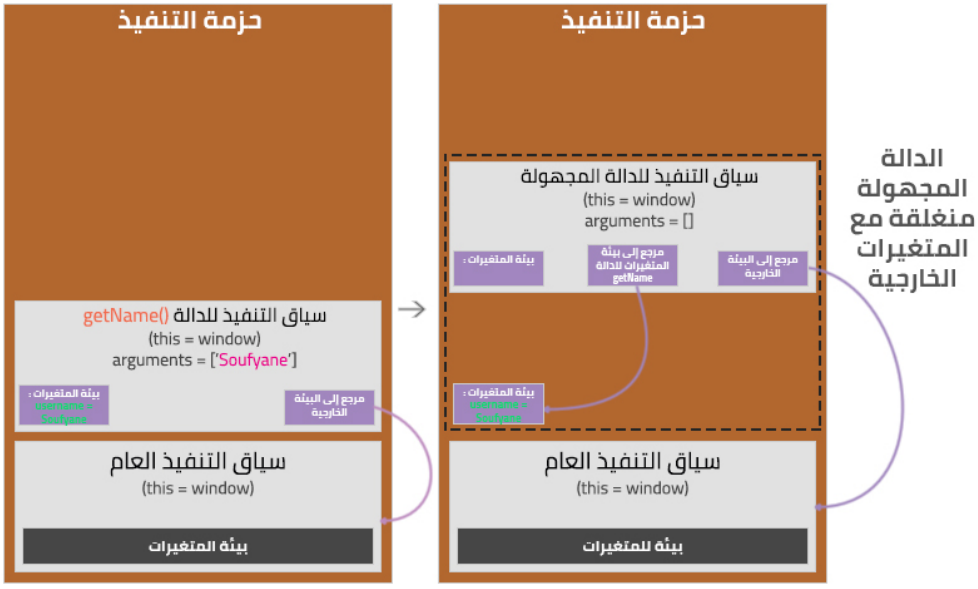
```
function getName(username){
    return function(){
        console.log('welcome ' + username);
    }
}

var uName = getName('Soufyane'); // ستَرجع دالة ثم تخرج من حزمة التنفيذ getName

uName();
```



كما ترى في هذا المثال أننا عرّفنا دالة getName، و التي تأخذ وسيطا واحدا، كل ما تقوم به هو إرجاع دالة مجهولة.
بعدها إستدعينا هذه الدالة، و مررنا لها السلسلة النصية 'Soufyane'، و قد أرجعت لنا الدالة التي يشير المتغير uName إلى موقعها في الذاكرة، و بهذا نستطيع أن نستدعيها ببساطة كما فعلنا في السطر التاسع uName() (يمكننا أن نستدعي الدالة المجهولة بشكل مباشر و ذلك كالتالي: getName()()).
هنا أريدك أن تتمعن جيدا فيما يلي:
بعد إستدعاء الدالة getName سترجع لنا هاته الدالة دالة مجهولة ثم تنتهي، و لو رأينا في النتيجة نجد أن الدالة المجهولة طبعت لنا قيمة الوسيط الممر للدالة السابقة، و التي إختفت بعد أن أدت عملها. إذن كيف حصلت هذه الدالة المجهولة على قيمة هذا الوسيط؟ لنعرف هذا علينا أن نعود إلى كومة التنفيذ، و نرى ما يحدث هناك:

نعرف أنه عند تنفيذ أي دالة يُنشأ لها سياق تنفيذ جديد، و الذي يملك مرجعا لمساحة في الذاكرة خاصة بالمتغيرات المعرّفة في الدالة، و منها الوسائط الممررة للدالة، و يملك أيضا مرجعا إلى البيئة الخارجية أين يبحث عن المتغيرات الغير معرفة داخل هاته الدالة، بعد إكتمال تنفيذ ما بداخل الدالة سيختفي سياقها مباشرة، و لكن هنا السؤال المطروح:
ما الذي يحدث لمساحة المتغيرات تلك التي في الذاكرة بعد أن يختفي سياق التنفيذ؟
حسنا في العادة، يقوم محرك جافا سكريبت بتنظيف هاته المساحات العالقة في الذاكرة عبر عملية تدعى بجمع القمامة (garbage collection)، و هي عملية معقدة نوعا ما.
إلا أنه في حالة الدوال المرجعة من دوال أخرى فإنه في هذه الحالة سيضمن محرك جافا سكريبت أن تتمكن تلك الدوال المرجعة من الوصول إلى مساحة المتغيرات الخاصة بالدوال التي أرجعتها، و لن تمس عملية جمع القمامة مساحة المتغيرات العالقة.
إذن بعد تنفيذ الدالة getName سيحدث كل ما سبق في حزمة التنفيذ، يأتي الآن المحرك لتنفيذ ما بداخلها فيجد جملة لإرجاع دالة مجهولة، يرجع هاته الدالة المجهولة، و بعدها يختفي سياق التنفيذ الخاص ب getName برمته إلا بيئة متغيراتها فإنها تبقى و لا تحذف و ذلك حتى تتمكن الدالة المجهولة من الوصول إلى بيئة المتغيرات تلك. بعد تنفيذ الدالة المجهولة فإنها تبحث عن المتغير username في بيئتها الخاصة بالمتغيرات فلا تجده فتنتقل بحثا عنه في بيئة المتغيرات الخارجية، و التي تملك مرجعا لها. و بهذا ستتمكن الدالة المجهولة من الوصول إلى المتغير usrname، و تطبع لنا قيمته.

الدوال المغلقة



هناك مثال آخر شائع في الإنترنت يشرح ماهية الدوال المغلقة، فلنلقي عليه نظرة من أجل زيادة الفهم:

```
function closurTest(){
    // إنشاء مصفوفة فارغة
    var arr = [];

    for (var i = 0; i < 3; i++){
        // arr إضافة دالة مجهولة كعنصر إلى المصفوفة
        arr.push(function(){
            console.log(i);
        });
    }
    return arr;
}

var result = closurTest();
console.log(result);
```

```
▼ (3) [ƒ, ƒ, ƒ]                                    app.js:15
  ▶ 0: ƒ ()
  ▶ 1: ƒ ()
  ▶ 2: ƒ ()
    length: 3
  ▶ __proto__: Array(0)
```

عرّفنا دالة تقوم بإنشاء مصفوفة فارغة ثم تضيف إليها 3 دوال مجهولة و ذلك عبر الوظيفة push()، و في الأخير ترجع لنا المصفوفة النهائية، بعدما قمنا بإستدعائها و تخزين النتيجة في المتغير result قمنا بطباعتها في وحدة التحكم، و النتيجة كما ترى. إذن يمكننا إستدعاء عناصر المصفوفة، و التي هي عبارة عن دوال مجهولة و ذلك كالآتي:

```
// إستدعاء الدوال المخزنة بالمصفوفة arr
result[0]();
result[1]();
result[2]();
```

و الآن ماهي النتيجة التي ستطبع في وحدة التحكم؟ هل هي : 0، 1 و 2؟ أم شيئ آخر؟ فكر بالأمر!

حسنا النتيجة قد تدهشك:

```
3                                   app.js:8
3                                   app.js:8
3                                   app.js:8
> |
```

3؟ من أين أتى بهذه القيمة؟؟

لنفهم الأمر خطوة، خطوة:

أولا لو تدقق في الشيفرة، فإنك تجدها تقوم بإضافة دوال مجهولة إلى مصفوفة فارغة حيث تقوم هذه الدوال بطباعة قيمة فقط. ثم ترجع تلك المصفوفة، هذا كل ما في الأمر. أما عن القيمة 3 فعندما يصل العداد i في الحلقة for إلى القيمة 3 يختل الشرط فتكسر الحلقة، و قد حصلت الدوال المجهولة على قيمة i بسبب أنها دوال منغلقة، حيث ضمن محرك جافا سكريبت لهذه الدوال المجهولة أن تصل للمتغير i رغم أن سياق التنفيذ الخاص بالدالة closurTest قد إختفى من كومة التنفيذ، و طباعتها للقيمة 3 سببه إستدعائها بعد أن تم إرجاع المصفوفة، أما في حالة إستدعائها داخل الحلقة فالنتائج ستختلف. هذا يظهر فائدة الدوال المنغلقة.

إذن كخلاصة للموضوع فإن أي دالة يتم تنفيذها، سيضمن لها المحرك أن تتمكن من الوصول إلى المتغيرات التي يُفترض أن تصل إليها، و هو ما يحصل مع الدوال المُرجعة؛ حيث تعتبر هذه ميزة في لغة جافا سكريبت، و لسنا بحاجة إلى أن نفعل ذلك بأنفسنا و ليس هناك كلمة مفتاحية لتوفر لنا هذه الميزة، بل هي أمر ضمني في اللغة.

فلسفة الكائنات في جافا سكريبت:

لقد رأينا كيف تمثل جافا سكريبت الكائنات بأسلوبها الخاص و ذلك عبر تعريف الكائن إما عن طريق القوسين المعقوفين { } أو عبر الدالة Object()، و تتيح لنا جافا سكريبت إضفاء الصفات (الخصائص) و الأفعال (الوظائف) على هذا الكائن بإستخدام أسلوب "خاصية: قيمة" (property: value)، أما بالنسبة للوراثة و هي تمكن كائن من الوصول إلى خصائص، و وظائف كائن آخر كأبسط تعريف، فإن جافا سكريبت تتبع منهجا مغايرا لأغلب لغات البرمجة كالجافا و السي# مثلا

حيث تعتمد فلسفة النماذج الأولية "prototypes" على عكس فلسفة الأصناف أو الأنواع بالنسبة لجافا مثلا و التي لها إمكانية أن تكون خاصة "private" أو عامة "public" أو محمية "protected"، و ما إلى ذلك من المفاهيم التي يجب أن تُلمّ بها لتتمكن من التعامل مع الأصناف في مثل هذه اللغات. لكن في جافا سكريبت لن تحتاج لأن تتعامل مع الأمور سابقة الذكر، بل كل ما تحتاجه هو فهم هذه فلسفة النماذج الأولية، و كيفية عملها، و ستدرك مدى روعة جافا سكريبت.

النموذج الأولي Prototype:

يعني مفهوم النموذج الأولي ذلك القالب الذي يصمم مبدئيا لأي منتوج حيث سيحصل المنتوج على نفس مواصفات و وظائف التصميم المبدئي، و بإسقاط هذا المفهوم في جافا سكريبت فإن النموذج الأولي لجميع الكائنات هو الكائن Object.prototype، حيث ترث منه جميع الكائنات خصائصه، و وظائفه. و تضمن جافا سكريبت الوراثة من الكائن Object عبر كائن يدعى prototype إذ يوجد هذا الكائن في جميع الكائنات بشكل مبدئي. أي أن النموذج المبدئي لأي كائن يكون كالتالي:

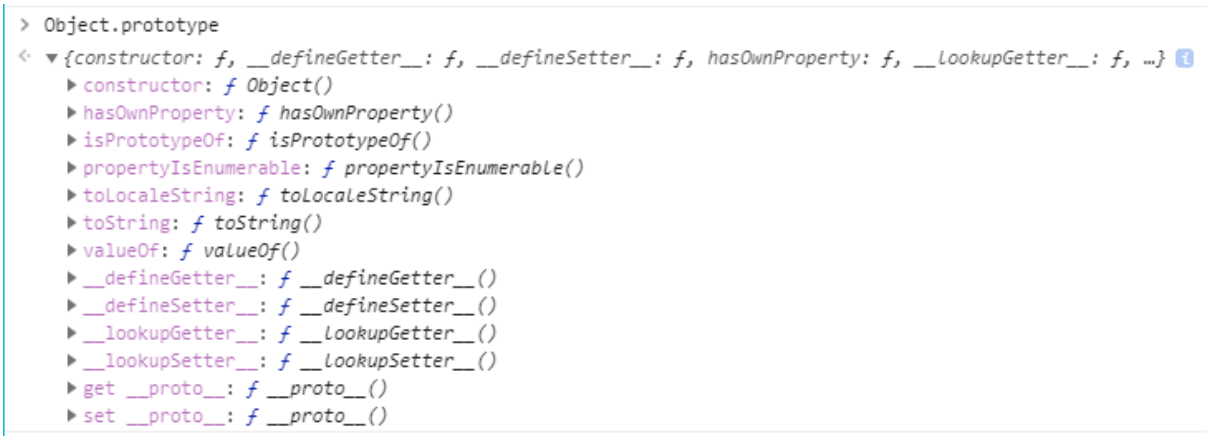
```
> var obj = {};
< undefined
> obj
< ▼{} i
    ▶ __proto__: Object
>
```

كما ترى أنشأنا كائنا فارغا، و لكنه يحتوى على خاصية مخفية ألا و هي كائن prototype الذي تحدثنا عنه، شاهد هنا أن إسم هذا الكائن في متصفح كروم __proto__، قد لا تجد هذا الإسم في متصفحات أخرى، مثلا متصفح موزيلا:

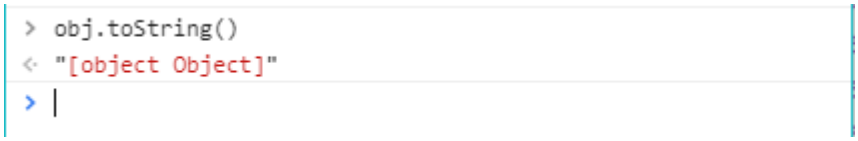
```
>> var obj = {};
← ▼{}
    | ◀ <prototype>: Object { _ }
>>
```

و كما قلنا فإن prototype عبارة عن كائن:

```
> typeof Object.prototype
< "object"
>
```



لاحظ أن هذا الكائن يحمل وظائف لم نعرّفها نحن، لكنها متوفرة بشكل إفتراضي. هذه الوظائف تابعة للكائن Object، و قد ورثها الكائن الجديد obj عن طريق "prototype"؛ إذ يمكننا أن نصل إلى تلك الوظائف مباشرة من الكائن obj و هذا صالح بالطبع لجميع الكائنات، فكما قلنا أنها جميعا ترث من الكائن Object:



لاحظ أننا لم نضطر إلى كتابة obj.__proto__.toString() بل إستدعينا الوظيفة مباشرة. و ذلك لأن عملية الوصول تتم بشكل ديناميكي.

و الأن لنقم بتخصيص نموذج مبدئي للكائن obj:

تنبيه:

ما سأقوم به هنا أمر غير مستحسن بتاتا لما له من تأثير على الأداء و بسبب أنه أهمل في الإصدارات الأخيرة، و لكني أستعمله من أجل التوضيح فقط لأن المتصفحات الحديثة تمكننا من الوصول إلى النموذج المبدئي مباشرة و هذا يساعدنا لفهم آلية عمل الوراثة في الجافا سكريبت، لاحقا سأشرح الطرق المثلى للوراثة، إلى ذلك الحين دعنا ندخل في الموضوع:

```
var obj = {};

var ourPrototype = {
    addProperty: function(key,value){
        obj[key] = value;
        console.log('Done');
    },

    greeting: function(){
        return 'hello dear!';
    }
}

// !!! لا تفعل هذا في المشاريع الفعلية
// !!! هذا من أجل التوضيح فقط
obj.__proto__ = ourPrototype;
```

قمنا في هذا المثال بتعريف كائن آخر بسيط ourPrototype، سيمثل النموذج المبدئي للكائن obj حيث سيرث منه جميع خصائصه و وظائفه عبر الكائن __proto__، (لاحظ الشرطتين السفليتين في بداية و نهاية إسم الخاصية و هذا أمر متعمّد لضمان عدم كتابتك لهذا الإسم خطأً، فأنت لن تلجأ إلى تسمية المتغيرات أو الخصائص في الكائنات بهذا الشكل مثلا!).

لنتأكد مما سبق دعنا نتحقق من ذلك في وحدة التحكم:

```
> obj
{} i
  __proto__:
    addProperty: ƒ (key,value)
    greeting: ƒ ()
    __proto__: Object
>
```

أنظر هنا، لقد تغيّر النموذج المبدئي للكائن obj، و هذا معناه أنه يمكننا الوصول إلى وظائف الكائن ourPrototype:

```
> obj.greeting();
"hello dear!"
>
```

```
> obj.addProperty('id', 112);
  Done                                    app.js:6
< undefined
> obj
< ▶ {id: 112}
>
```

شاهد ماذا حدث عندما إستدعينا الوظيفة addProperty المعرفة داخل الكائن ourPrototype، و الذي يعد النموذج المبدئي لـ obj، تقوم هذه الوظيفة بإضافة مفتاح-قيمة إلى الكائن، و بما إننا إستدعيناها على الكائن obj فقد أضافت الزوج (المفتاح-القيمة) إليه بدلا من الكائن الأصلي الذي يحتويها، و هذا لأن المتغير this في هذه الحالة يشير إلى الكائن الذي أُستدعِيت عليه الوظيفة حتى لو لم تكن معرفة عليه، المهم أنه يستطيع الوصول إليها. ستفيدنا هاته الميزة لاحقا عند إنشاء الكائنات عن طريق المعامل new.

و الآن أريد أن أنبهك لأمر ما: أنظر في وحدة التحكم بالنسبة للكائن ourPrototype الذي يعتبر نموذجا مبدئيا للكائن obj؛ ألا ترى أن له الخاصية __proto__ أيضا؟ و التي تشير إلى النموذج المبدئي للكائن ourPrototype، إذن ما هو هذا النموذج في هذه الحالة؟ لنقم بتوسيعه و نرى ماهيته:

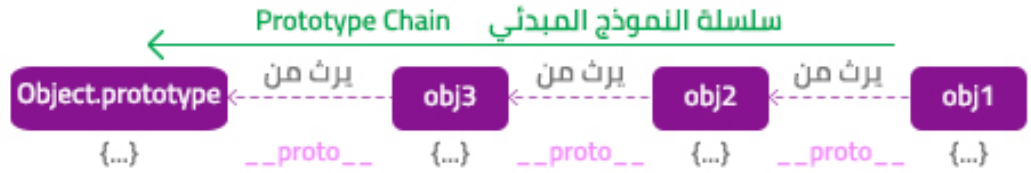
إذن كما ترى فإنه يشير إلى الكائن Object.prototype، و لو عدنا لما قلناه في البداية أن كل كائن يرث من هذا الكائن الرئيسي، فهذا يعني أن سلسلة وراثة نشأت هنا، فـ obj يرث من ourPrototype و هذا الأخير يرث من الكائن الرئيسي Object.prototype، و هو آخر نقطة في هذه السلسلة، و منه فإن عملية الوصول إلى خصائص كائن ما تتم عبر عملية بحث تبتدأ أولا بالبحث في الكائن ذاته، إن وجدت يتم إرجاع تلك الخاصية، أو تنفذ تلك الوظيفة مباشرة، فإن لم توجد

يتم البحث في سلسلة النموذج المبدئي (prototype chain) إلى أن يجدها ليرجعها، أو ينفذها مباشرة، فإن لم توجد يكمل البحث حتى آخر نقطة في السلسلة، و هي الكائن Object.prototype.

فمثلا عند محاولة الوصول إلى الوظيفة ()toString من خلال الكائن obj فإن المحرك أولا سيبحث عنها في هذا الكائن، فلا يجدها، و في هذه الحالة لا يرجع خطأ مباشرة، بل يواصل البحث في سلسلة النموذج المبدئي، فيبحث في الكائن ourPrototype، و لا يجدها أيضا، ثم يواصل البحث في النموذج المبدئي للكائن ourPrototype، و هو الكائن الرئيسي Object.prototype فيجدها هناك، و بالتالي يقوم بتنفيذ هذه الوظيفة بشكل طبيعي.
هذه العملية تتم دائما عند الوصول إلى خصائص، أو وظائف أي كائن، و في حال عدم وجودها لا في الكائن و لا في سلسلة النموذج المبدئي، يرجع لنا المحرك القيمة undefined بالنسبة للخصائص، و خطأ بالنسبة للوظائف لأنه سيحاول تنفيذ undefined في هذه الحالة:

```
> obj.name
< undefined
> obj.getName()
✖ ▸Uncaught TypeError: obj.getName is not a function    VM1391:1
      at <anonymous>:1:5
>
```

قد تتشابه سلسلة النموذج (Prototype Chain) مع سلسلة النطاق (Scope Chain) إلا أنهما مختلفان تماما، فالأولى تُعنى بالبحث عن خصائص، و وظائف عبر سلسلة من الكائنات المتصلة عبر خاصية prototype، أما الثانية فتُعنى بمكان الوصول إلى المتغيرات.



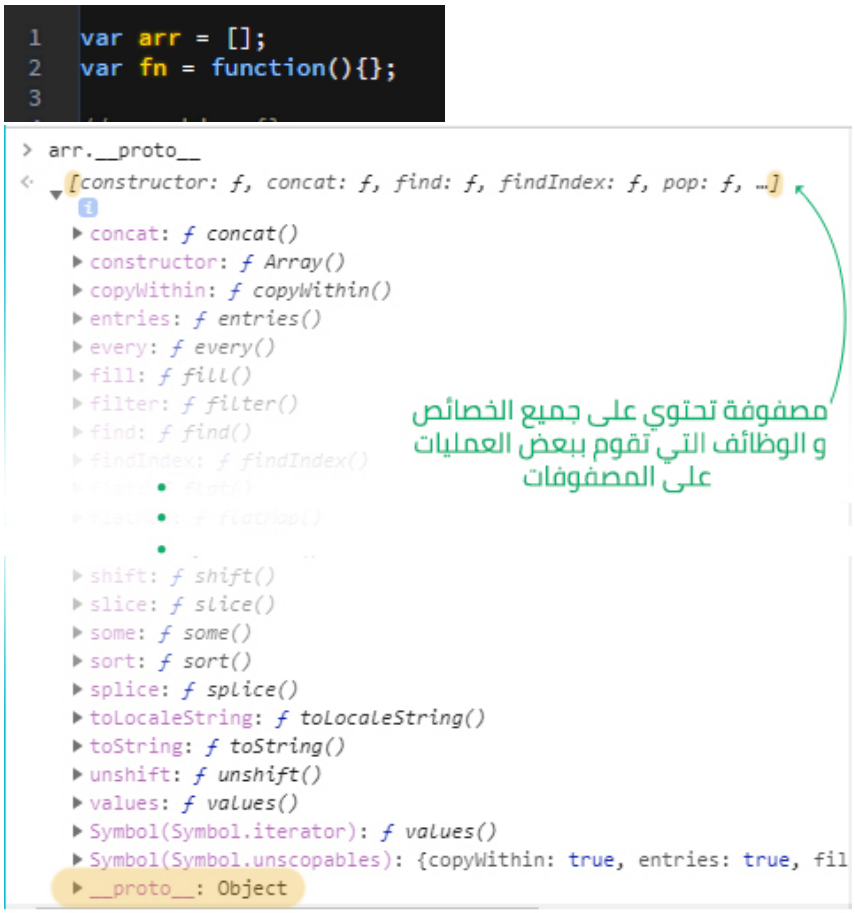
لاحظ أن هذه السلسلة يجب أن تكون في إتجاه واحد دائما، و لا يمكن أن تكون حلقية، مثال:

```
14  // !!! لا تستعمل __proto__ في المشاريع الفعلية !!!
15  // !!! هذا من أجل التوضيح فقط !!!
16  obj.__proto__ = ourPrototype;
17
18  ourPrototype.__proto__ = obj;
19
```

```
✖ ▸Uncaught TypeError: Cyclic __proto__ value                app.js:17
      at Object.set __proto__ [as __proto__] (<anonymous>)
      at app.js:17
>
```

حاولنا هنا أن نجعل كل كائن نموذج مبدئي للآخر، و قد أعطانا هذا خطأ.

يمكننا أن نجعل عدة كائنات ترث من نفس الكائن عبر جعل نموذجها المبدئي يشير إلى الكائن المنشود، و هذا ما يحدث في الأصل مع جميع الكائنات إذ أنها ترث من الكائن الرئيسي Object.prototype.
و الآن لنتأكد من هذا القول مع الدوال، و المصفوفات؛ إذ بإعتبارها كائنات أيضا:



لاحظ أن النموذج المبدئي للمصفوفة التي عرّفناها في هذا المثال، و لجميع المصفوفات يشير إلى مصفوفة مبدئية تحمل جميع الخصائص، و الوظائف التي كنّا نستعملها لبعض العمليات على المصفوفات، من بينها الخاصية length و الوظيفة join ... الخ. هاته المصفوفة في حد ذاتها كائن، عند توسيع تلك المصفوفة ستلاحظ في الأخير وجود الخاصية المخفية __proto__، و التي تعني النموذج المبدئي لهذا الكائن بدوره، و كما ترى فإنها تشير إلى الكائن الرئيسي Object.prototype:

```
> arr.__proto__.__proto__
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnP
roperty: ƒ, __lookupGetter__: ƒ, …}
  constructor: ƒ Object()
  hasOwnProperty: ƒ hasOwnProperty()
  isPrototypeOf: ƒ isPrototypeOf()
  propertyIsEnumerable: ƒ propertyIsEnumerable()
  toLocaleString: ƒ toLocaleString()
  toString: ƒ toString()
  valueOf: ƒ valueOf()
  __defineGetter__: ƒ __defineGetter__()
  __defineSetter__: ƒ __defineSetter__()
  __lookupGetter__: ƒ __lookupGetter__()
  __lookupSetter__: ƒ __lookupSetter__()
  get __proto__: ƒ __proto__()
  set __proto__: ƒ __proto__()
>
```

نفس الأمر بالنسبة للدوال حيث أن نموذجها المبدئي يشير إلى دالة فارغة:

```
> fn.__proto__
ƒ () { [native code] }
> fn.__proto__.apply
```

apply
arguments
bind
call
caller
constructor
length
name
toString
hasOwnProperty Object
isPrototypeOf
propertyIsEnumerable
toLocaleString
valueOf
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__

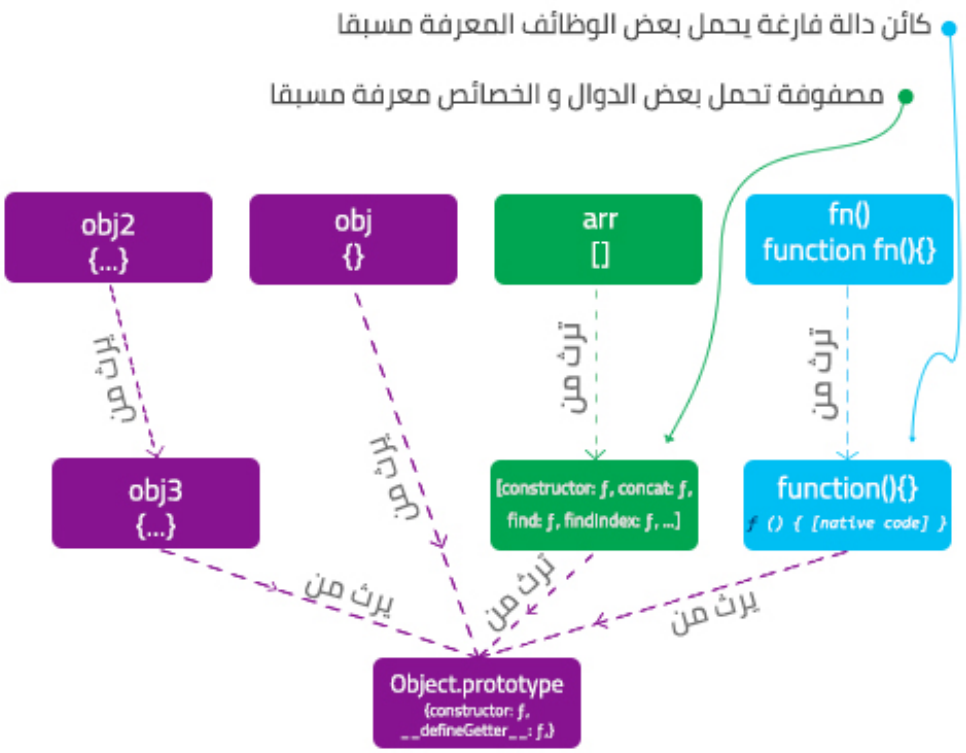
الكلمات المظللة بالبرتقالي
تابعة لكائن الدالة
المبدئي

هل تتذكر الخاصية name التي تحدثنا عنها من قبل، و الوظائف الثلاثة apply، bind و call، كلها معرّفة مسبقا في كائن الدالة المبدئي. جميع الدوال ترث من هذا الكائن، أمّا بقية الخصائص أو الوظائف فإنها موروثة عن الكائن الرئيسي Object.prototype:

لتتأكد أن الكائن Object.prototype هو آخر كائن في سلسلة النماذج المبدئية. حاول الوصول إلى النموذج المبدئي لهذا الكائن، لترى أنك ستحصل على القيمة null و التي تعني لا شيئ في الجافا سكريبت:

و هذا توضيح يلخص ما قلناه سابقا:

الوراثة في جافا سكريبت



حسنا، ماذا لو أردنا أن يرث كائن واحد من عدة كائنات؟

للأسف هذه الخاصية لا تدعمها ميزة النموذج المبدئي prototype و لكن هناك طريقة يتبعها أغلب المبرمجين ألا وهي النسخ، ليس يدويا بالطبع و لكن برمجيا، حيث يتم إنشاء دالة تقوم بأخذ عدة كائنات كمعاملات ثم تقوم بنسخ خصائصها، و وظائفها إلى الكائن الهدف. هذه الطريقة تدعى بالتوسعة extending و ستجدها في أكبر المكتبات، و أطر العمل كـ jQuery، و underscore ..الخ. لنأخذ مثالا من مكتبة underscore نظرا لبساطتها حتى نوضح فيه هذه الطريقة:

```
1074  // An internal function for creating assigner functions.
1075  var createAssigner = function(keysFunc, defaults) {
1076    return function(obj) {
1077      var length = arguments.length;
1078      if (defaults) obj = Object(obj);
1079      if (length < 2 || obj == null) return obj;
1080      for (var index = 1; index < length; index++) {
1081        var source = arguments[index],
1082            keys = keysFunc(source),
1083            l = keys.length;
1084        for (var i = 0; i < l; i++) {
1085          var key = keys[i];
1086          if (!defaults || obj[key] === void 0) obj[key] =
          source[key];
1087        }
1088      }
1089      return obj;
1090    };
1091  };
1092
1093  // Extend a given object with all the properties in passed-in
     object(s).
1094  _.extend = createAssigner(_.allKeys);
1095
```

أولا أريد أن أخبرك بشيء و هو: إياك أن تخاف من تصفح الشيفرة المصدرية لأي مكتبة كانت، فهي مجرد أكواد كتبها بشر ليسوا بخارقين، و لم يولدوا متعلمين! بل كل ما يميّزهم هو أنهم قد تمكنوا من المفاهيم المتقدمة في جافا سكريبت، و طبقوا تلك المهارات التي إكتسبوها.
هذا كل ما في الأمر. و لو فهمت كل ما تم شرحه في هذا الكتاب، فأهنئك على هذا، و أبشرك أنك على إستعداد لكتابة مكتبتك الخاصة و بنفسك! فقط يجب أن تتمرن، فالممارسة تولّد الإمتياز "practice makes perfect!". و كما إستطاعوا هم فأنت أيضا تستطيع.

لنبدأ الشرح:
أولا: تمّ تعريف الوظيفة extend التابعة للكائن _، كما في السطر 1094. تشير هذه الوظيفة إلى الدالة المرجعة من الدالة createAssigner()، حيث تأخذ createAssigner وسيطين، الوسيط الثاني لا يهمنا في هذه الحالة، كل ما تقوم به هذه الدالة هي إرجاع دالة مجهولة أرأيت؟ إنهم يستعملون ميزة الدوال المنغلقة التي شرحناها من قبل. تستقبل هذه الدالة المجهولة عدة كائنات كوسائط بالرغم من أنه تم تعريف معامل واحد obj!
السر في إستعمال شبه المصفوفة arguments الذي يمكّننا من الوصول إلى جميع الوسائط الممرّرة إلى الدالة. و هذا لأن الدالة المجهولة مصممة لتأخذ أكثر من معامل (كائن) لتنسخ خصائصهم و تضيفها إلى الوسيط الأول (الكائن).
السطر الذي يليه للتحقق من الوسيط الثاني defaults لا يهمنا في هذه الحالة.
بعده يتم التحقق من عدد الكائنات الممرّرة للدالة المجهولة، فإن كان أقل من 2 فهذا يعني أن كائنا واحدا قد مُرِّر أو لم يمرّر أي شيئ، و بالتالي يتم إرجاع ذاك الكائن، أما إذا لم يمرر أي شيئ

فسيتم إرجاع undefined و هذا منطقي، فإذا مرّرت للدالة extend كائنا واحدا سترجعه لك كما هو، و لن يتم إضافة أي شيئ.

في حالة مُرر أكثر من كائن:
تم تعريف حلقة تبدأ العد من 1 أي العنصر الثاني في شبه المصفوفة arguments، و هذا واضح لأننا سنمرّر الكائن المراد توسعته أولا، و بالتالي سيكون العنصر الأول في arguments أي فهرسه 0.
بعدها تم تعريف 3 متغيرات:
source: لتخزين كل كائن مؤقتا.
keys: لتخزين مفاتيح الكائن المخزن في source داخل المصفوفة. تم تعريف وظيفة تدعى allKeys حيث تأخذ كائنا كوسيط لتقوم بتخزين كل المفاتيح التي يحملها داخل مصفوفة فارغة، ثم ترجع المصفوفة النهائية كنتيجة. لذلك تم تمريرها للدالة createAssigner مكان المعامل keysFunc.
(أنظر السطر 1094)
l: يخزن طول المصفوفة keys.
بعدها تم تعريف حلقة ثانية داخل الحلقة الأولى، و وظيفتها المرور على جميع المفاتيح المخزنة في المصفوفة keys لإضافتها إلى الكائن الممرّر كأول وسيط (obj)، و يتم ذلك عبر تخزين كل مفتاح مؤقتا في المفتاح key، ثم يتم التحقق من أن هذا المفتاح غير موجود سابقا في ذاك الكائن و هذا عن طريق السطر 1086 الذي إن كانت قيمة defaults! تساوي true، أو قيمة التعبير obj[key] === void 0 تساوي true، و void 0 ترجع لنا القيمة undefined أي أن المفتاح الحالي في الكائن غير موجود، إذاً، في حال تتحقق الشرط سيتم إضافة ذاك المفتاح إلى الكائن الأول، و تسند له القيمة الموافقة في الكائن source. أتذكر! الكائنات تتعامل بالمرجعية.
تعاد العملية مع بقية المفاتيح إلى أن تنتهي الحلقة، و بالتالي ينتقل إلى الكائن الثالث، و الرابع، و هكذا على حسب عدد الكائنات الممرّرة للدالة extend.
الشرح كبير، لكن الفكرة عندما تُفهم ستبدو بسيطة جدا. حيث إستطعنا أن نجعل كائنا يتشارك خصائص، و وظائف كائنات أخرى عبر النسخ، أو لنقل إمتلاك عناوينها المرجعية.
هنا لا يتم تخصيص النموذج المبدئي، و إنما يتم نسخ الخصائص بالمرجعية، و تضمينها في كائن محدد، و بالتالي توسيعه.

هل رأيت كم هذا سهل جدا، لقد إطّلعت على جزء من الشيفرة المصدرية لمكتبة underscore، و تعلمت منها شيئا، الأمر ليس بتلك الصعوبة التي كنت تتخيلها، أليس كذلك؟

كتنويه فقط، ستتوفر النسخة القادمة من جافا سكريبت على دالة تدعى extands مع وجود حرف s في الأخير حيث ستستعمل لتحديد النموذج المبدئي.

إنشاء الكائنات:

إلى الآن بعد أن فهمنا كيف تتم الوراثة في جافا سكريبت، و ذلك عبر خاصية النموذج المبدئي، و بعد أن حذرنا من إستعمال الخاصية __proto__ (يجدر التنويه أيضا إلى عدم إستعمال الوظيفة setPrototypeOf التابعة للدالة Object التي تعمل على تغيير النموذج كما __proto__) و ذلك بأن تغيير النموذج المبدئي بعد إنشاء الكائن، له تأثير سلبي على أداء التطبيق، حان الوقت لنستعمل الطرق المثلى، و ذلك بتغيير النموذج المبدئي أثناء إنشاء الكائن، و للقيام بهذا نستعين بإحدى الطريقتين اللتين قد تكلمنا عنهما في السابق في مطلع فصل الكائنات، ألا و هما:

- إنشاء الكائنات عن طريق Object.create
- بإستخدام المعامل new مع الدوال.

إنشاء الكائنات عن طريق Object.create[3]:

تستقبل هذه الوظيفة وسيطين، أولهما هو النموذج المبدئي، و الذي يكون إما كائنا، أو null، و الوسيط الثاني هو الكائن، أو لنقل خصائص الكائن الجديد الذي نحن بصدد إنشائه، و هو إختياري.
لنأخذ مثالا للتوضيح:

تُستخدم الوظيفة create التابعة للدالة Object لتغيير النموذج المبدئي أثناء إنشاء الكائن، و هذا يدخل ضمن أفضل المهارات:
أولا دعنا ننشئ كائنا ليكون النموذج المبدئي لـ obj (أي يرث منه) و لنسمّه myPrototype:

---

[3] تتوفر هذه الوظيفة في المتصفحات الحديثة

```
var myPrototype = {

    greet: function(){
        console.log('Hello ' + this.name);
    },

    printFullName: function(){
        console.log(this.name + ' ' + this.lastName);
    }
};

var obj = Object.create(myPrototype)

```

```
> obj
{} 
  __proto__:
    greet: ƒ ()
    printFullName: ƒ ()
    __proto__: Object
```

كما ترى في هذا المثال، أنشئنا الكائن myPrototype و الذي إستخدمناه كنموذج مبدئي للكائن obj المنشئ بواسطة الطريقة create()، و عندما تحققنا في وحدة التحكم، و جدنا أنا الكائن obj قد ورث من الكائن myPrototype أي أن هذا الأخير أصبح نموذجا مبدئيا للكائن obj.

يمكننا أن نضيف الخصائص ل obj بشكل عادي:

```
var obj = Object.create(myPrototype)

obj.name = 'Soufyane';
obj.lastname = 'Hedidi';

```

```
> obj
{name: "Soufyane", lastname: "Hedidi"}
  lastname: "Hedidi"
  name: "Soufyane"
  __proto__:
    greet: ƒ ()
    printFullName: ƒ ()
    __proto__: Object
```

الأمر عادي كما ترى فقد تمت إضافة الخاصيتين بعد إنشاء الكائن بشكل طبيعي، لكن أليس من الأفضل أن نضيف الخصائص أثناء إنشاء الكائن؟ بلى هذا ممكن، و هنا يأتي دور الوسيط الثاني الخاص بالطريقة create() و الذي يحتاج إلى شيئ من التفصيل.

سابقا عند إنشائنا لكائن بالطريقة الحرفية، كنا نستطيع عد خصائصه، و تحديث قيمها، أو حتى حذفها، لكن هل يمكننا التحكم في كل هذه العمليات من تحديث، و عد، و حذف للخاصية بأكملها حيث نجعلها ممكنة، أو العكس؟

لحسن الحظ نعم، يمكننا فعل ذلك، و هذا من خلال الوسيط الثاني -الإختياري- الذي يُمرَّر إلى الوظيفة create() حيث يكون عبارة عن كائن؛ إذ يعمل بمثابة واصف للخصائص الجديدة التي

ستُعرّف أو ستضاف إلى الكائن المُنشأ، يحتوي هذ الكائن الواصف على واصفات، إما تصف بيانات الخاصية، أو تصف الوصول إليها.

واصفات البيانات:

يقصد بها قيمة الخاصية، و إمكانية تحديثها. و هي إختيارية، يمكنك تحديدها حسب حاجتك.

1. **value : (القيمة)**

   تحمل القيمة التي ستُعطى للخاصية. و قيمتها الإفتراضية undefined.

2. **writable : (قابلية الكتابة)**

   تسمح بتغيير، أو تحديث قيمة الخاصية. تأخذ إحدى القيمتين المنطقيتين true لإمكانية التحديث، و false لمنع ذلك، قيمتها الإفتراضية هي false.

واصفات الوصول:

1. **set: (الواضع)**

   هي دالة تُنَفّذ أو تُستدعى كلما تم إسناد قيمة إلى الخاصية، حيث تمكّن من وضع تلك القيمة للخاصية، إذن هي تعمل كواضع لقيمة الخاصية، و لهذا سميت بالواضع. القيمة الإفتراضية لهذه الواصفة هي undefined.

2. **get: (الجالب)**

   هي دالة تنَفّذ كل ما تم الوصول إلى الخاصية، فهي تعمل كجالب لقيمة الخاصية، قيمة هاته الواصفة الإفتراضية هي undefined.

عليك أن تعلم أنه لا يمكنك الجمع بين واصفات البيانات و واصفات الوصول. يمكنك إستخدام إحداهما فقط، و إلا ستحصل على خطأ. سنوضح هذا بالأمثلة القادمة.

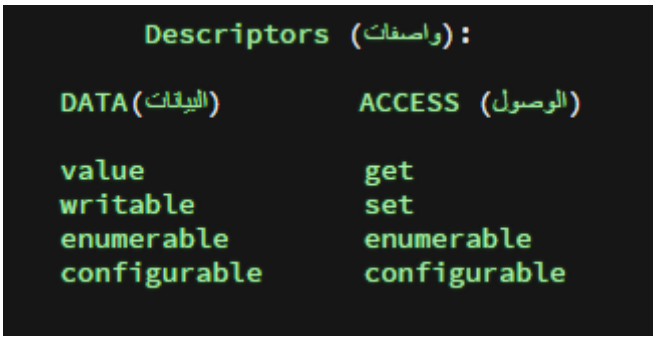
تتشارك كل من واصفات البيانات و واصفات الوصول هاتين الواصفتين:

1. **enumerable: (قابلية العد)**

تعني إمكانية عد الخاصية أثناء عدّ خصائص الكائن الذي يحتوي عليها، و ذلك مع الحلقة for in ... أو الطريقة Object.keys() التي تعد خصائص الكائنات، و أيضا تحدد إمكانية إختيار الخاصية من قبل الطريقة assign() أو المعامل spread (مثلا arg...). تأخذ هذه الواصفة إحدى القيمتين المنطقيتين true لإمكانية العد، و false لمنع ذلك، و قيمتها الإفتراضية false.

2. **configurable: (قابلية الضبط)**

تسمح هاته الواصفة بإمكانية تعديل الواصفات، و حتى إمكانية حذف الخاصية أيضا إذا كانت قيمتها true. قيمتها الإفتراضية false.



لنشرح هاته الواصفات بشيء من التفصيل و بعض الأمثلة حتى تتضح الأمور:

```
11
12  var obj = Object.create(myPrototype, {})
13
```

```
> obj
◀ ▼{}
    ▼__proto__:
      ▶greet: ƒ ()
      ▶printFullName: ƒ ()
      ▶__proto__: Object
```

في هذا المثال مررنا كائنا فارغا كوسيط ثاني للوظيفة create()، و قد أنشأت لنا الكائن obj كما كان الأمر سابقاً عندما لم نمرر الوسيط الثاني.

و الآن لإضافة الخصائص أثناء الإنشاء فإنه يجب علينا تعريفها على شكل كائنات تحمل الواصفات التي تكلمنا عنها سابقا و إذا لم نبلغ عنها فإنها ستتخذ قيمها الإفتراضية:

```
var obj = Object.create(null ,{
                            name: {
                                // value: undefined,
                                // writable: false,
                                // enumerable: false,
                                // configurable: false
                            },
                            lastname: {
                                // value: undefined,
                                // writable: false,
                                // enumerable: false,
                                // configurable: false
                            }
                        });
```

```
> obj
◂ ▸{name: undefined, lastname: undefined}
```

كما ترى أنه تم تعريف الخاصيتين name و lastname داخل الكائن obj، و بما أننا لم نحدد أيا من الواصفات الأربعة فقد تم أخذ القيم الإفتراضية، و لهذا أسندت القيمة undefined إلى الخاصيتين، و لو أردنا أن نتحقق من هاته الواصفات فإننا نستعين بالوظيفة getOwnPropertyDescriptor() التابعة للدالة Object كالتالي:

```
> Object.getOwnPropertyDescriptor(obj, 'name')
◂ ▸{value: undefined, writable: false, enumerable: false, configurable: false}
> Object.getOwnPropertyDescriptor(obj, 'lastname')
◂ ▸{value: undefined, writable: false, enumerable: false, configurable: false}
```

لاحظ الواصفات و قيمها لكل خاصية، حيث نمرر الكائن أولا ثم الخاصية إلى الوظيفة getOwnPropertyDescriptor() لكي ترجع لنا واصفاتها.

و بالنسبة للخصائص المضافة عن طريق خاصية التنقيط، فإنها قابلة للتحديث و العد و الحذف مما يعني أن الواصفة value تأخذ القيمة المسندة للخاصية و بقية الواصفات تأخذ القيمة true شاهد المثال التالي:

```
> obj.age = 26;
< 26
> Object.getOwnPropertyDescriptor(obj, 'age');
< ▶ {value: 26, writable: true, enumerable: true, configurable: true}
> obj.age = 'twenty six old';
< "twenty six old"
> obj.age;
< "twenty six old"
> Object.getOwnPropertyDescriptor(obj, 'age');
< ▶ {value: "twenty six old", writable: true, enumerable: true, configurable: true}
> for (key in obj){
      console.log(key);
  }
  age
< undefined
> delete obj.age;
< true
>
```

والآن لنعدل في تلك الواصفات حتى نرى تأثيرها على الخصائص:

واصفة القيمة (value):

تتكفل هاته الواصفة بتعيين قيمتها كقيمة إلى الخاصية:

```
11
12 ▼ var obj = Object.create(null ,{
13 ▼                               name: {
14                                      value: 'Soufyane',
15                                     // writable: false,
16                                     // enumerable: false,
17                                     // configurable: false
18                                 },
19 ▼                               lastname: {
20                                      value: 'Hedidi',
21                                     // writable: false,
22                                     // enumerable: false,
23                                     // configurable: false
24                                 }
25                             });
```

```
> obj
{name: "Soufyane", lastname: "Hedidi"}
    lastname: "Hedidi"
    name: "Soufyane"
> Object.getOwnPropertyDescriptor(obj, 'name')
{value: "Soufyane", writable: false, enumerable: false, configurable: false}
> Object.getOwnPropertyDescriptor(obj, 'lastname')
{value: "Hedidi", writable: false, enumerable: false, configurable: false}
>
```

كما ترى في المثال أعلاه، تم تعيين قيمة المفتاح value كقيمة للخاصية.

واصفة قابلية الكتابة (writable):

```
var obj = Object.create(myPrototype ,{
                        name: {
                            value: 'Soufyane',
                            writable: true,
                            // enumerable: false,
                            // configurable: false
                        },
                        lastname: {
                            value: 'Hedidi',
                            // writable: false,
                            // enumerable: false,
                            // configurable: false
                        }
                    });
```

```
> obj
{name: "Soufyane", lastname: "Hedidi"}
> obj.name = 'Omar'
"Omar"
> obj.name
"Omar"
> obj.lastname = true
true
> obj.lastname
"Hedidi"
>
```

كما ترى، قمنا بتغيير قيمة الخاصية name بشكل عادي، و ذلك لأنها قابلة للكتابة (writable: true) أو التحديث، أما فيما يخص الخاصية lastname فإن قيمتها لم تتغير، و ذلك بسبب أنها غير قابلة للكتابة (writable: false).

في حال إستخدام الوضع الصارم ("use strict") فإنه سينتج خطأ (TypeError) عند محاولة تحديث قيمة الخاصية الغير قابلة للكتابة أو التعديل:

```
> 'use strict'
  obj.lastname = 123
✖ ▸Uncaught TypeError: Cannot assign to read only property 'lastname' of object '[object Object]'
      at <anonymous>:2:14
>
```

هناك أمر يجب أن تبقيه في ذهنك أيضا، و هو أن الخصائص الغير قابلة للكتابة و التي قيمها عبارة عن كائنات، لا يمكن تغيير إشارتها المرجعية، في حين أنه يمكن تغيير خصائص الكائنات التي تشير إليها. و المثال خير سبيل للفهم:

```
11
12 ▼ var obj = Object.create(null ,{
13 ▼                           ob: {
14                                 value: {a: 1}
15                             }
16                             });
17
```

```
> obj
  ▼{ob: {…}} i
    ▸ob: {a: 1}
> obj.ob = 20
  20
> obj.ob
  ▸{a: 1}
> 'use strict'
  obj.ob = 20
✖ ▸Uncaught TypeError: Cannot assign to read only property 'ob' of object '[object Object]'
      at <anonymous>:2:8
> obj.ob.a
  1
> obj.ob.a = false
  false
> obj.ob.a
  false
> |
```

أرأيت ذلك؟ لم نتمكن من تحديث قيمة الخاصية ob (أي أننا لم نتمكن من تغيير الإشارة المرجعية) و قد حصلنا على خطأ عندما إستخدمنا الوضع الصارم. في حين أننا تمكّنا من تحديث ما بداخل الكائن الذي تشير إليه الخاصية ob و هذا واضح طبعاً، لأنه كائن مستقل بحد ذاته، و خصائصه قابلة للتعديل، و العد أيضا! (سنعرف ما السبب فيما بعد). أما فيما يخص الخاصية ob فإن قيمتها هي إشارة مرجعية، و هي غير قابلة للتعديل. هنا مربط الفرس!

واصفة الجلب (get):

تُعرّف هاته الواصفة على شكل دالة تعمل بمثابة جالب للخاصية، حيث أن القيمة التي ترجعها الدالة تكون عبارة عن قيمة للخاصية، عند محاولة الوصول إلى الخاصية فإن هذه الدالة تستدعى

بلا أية معاملات، مع إشارة المعامل this إلى الكائن الذي تم إستدعاء الخاصية من خلاله (قد لا يكون هو الكائن الذي عُرّفت عليه و إنما ورثها)، في حال لم يتم تعريف دالة، فإن قيمة هاته الواصفة الإفتراضية هي undefined.

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                          name: {
14                                  get: undefined,
15                                  // set: undefined,
16                                  // enumerable: true,
17                                  // configurable: true
18                              }
19                          });
20
```

```
> obj
◂ ▼{} i
    name: <unreadable>
  ▶ __proto__: Object
> obj.name
◂ undefined
>
```

كما ترى في هذا المثال، تم تعريف الخاصية name داخل الكائن obj، و لكنها غير قابلة للقراءة <unreadable> حسب ما يظهر في السطر الأول في وحدة التحكم، أثناء محاولتنا الوصول إلى الخاصية name فإنه تم تنفيذ الواصفة get و بما أننا لم نعرّف بها أية دالة فإنها تأخذ القيمة الإفتراضية undefined و ترجعها لنا.

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                          name: {
14 ▼                                get: function(){
15                                      console.log('تم تنفيذ دالة
                                        الجلب');
16                                      return 'Ahmed';
17                                  },
18                                  // set: undefined,
19                                  // enumerable: true,
20                                  // configurable: true
21                              }
22                          });
23
```

```
> obj
◂ ▾{} i
      name: (...)
    ▸ get name: ƒ ()
    ▸ __proto__: Object
> obj.name
  تم تنفيذ دالة الجلب
◂ "Ahmed"
> obj.name === "Ahmed";
  تم تنفيذ دالة الجلب
◂ true
>
```

في هذا المثال عيّنا دالة للواصفة get حيث تقوم بطباعة جملة إلى وحدة التحكم لنتأكد بأنه تم تنفيذها ثم ترجع لنا السلسلة النصية 'Ahmed' إذ أن هاته القيمة المرجعة تصبح قيمة للخاصية name. و مهما حاولنا تغيير هاته القيمة عبر معامل الإسناد فإن القيمة ستبقى السلسلة النصية 'Ahmed':

```
> obj.name = 92;
◂ 92
> obj.name === "Ahmed";
  تم تنفيذ دالة الجلب
◂ true
>
```

قلنا سابقا أن المتغير this يشير إلى الكائن الذي تم الوصول إلى الخاصية من خلاله؛ إذ أنه يمكن يكون ورثها عبر سلسلة النموذج المبدئي، لنرى هذا في مثال يوضح ذلك:

```
1  var myPrototype = { name: 'myPrototype' };
2
3  Object.defineProperty(myPrototype, 'lastname', {
4      get: function(){
5          console.log('this => ' + this.name + ' :'); // => تعني يشر إلى
6          console.log(this);
7      }
8  })
9
10 var obj = Object.create(myPrototype); // obj إلى myPrototype توريث خصائص
11
12 obj.name = 'obj';
```

```
> obj
◂ {name: "obj"}
> myPrototype
◂ {name: "myPrototype"}
> obj.lastname
  this => obj :
  {name: "obj"}
◂ undefined
> myPrototype.lastname
  this => myPrototype :
  {name: "myPrototype"}
◂ undefined
>
```

كما ترى في هذا المثال، يشير المعامل this إلى الكائن الذي استُدعيَت من خلاله الخاصية، فالخاصية lastname معرفة على الكائن myPrototype و الذي ورّثنا خصائصه إلى الكائن obj عبر الوظيفة create() و بالتالي فإن obj يستطيع الوصول إلى الخاصية lastname عبر سلسلة النموذج المبدئي، حينما نصل إلى هذه الخاصية عبر obj فإن المعامل this يشير إليه، أما عندما نصل إلى lastname عبر الكائن myPrototype فإن this يشير إلى هذا الأخير.

واصفة الوضع (set):

تُعرّف هاته الواصفة على شكل دالة تعمل بمثابة واضع لقيمة الخاصية، حيث تُستدعى هذه الدالة عندما يتم إسناد قيمة للخاصية عن طريق معامل الإسناد، و تُمرَّر تلك القيمة المسندة كمعامل لها، مع إشارة المعامل this إلى الكائن الذي تمّ إسناد قيمة للخاصية من خلاله. إذا لم تعرَّف دالة وضع فإن قيمة الخاصية الإفتراضية هي undefined.

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                             name: {
14 ▼                                 get: function(){
15                                      return this.val;
16                                  },
17                                  set: undefined,
18                                  // enumerable: true,
19                                  // configurable: true
20                              }
21                          });
22
```

```
> obj
{} i
    name: (...)
  ▶ get name: ƒ ()
  ▶ __proto__: Object
> obj.name = 'Soufyane';
"Soufyane"
> obj.name
undefined
>
```

كما يظهر في المثال، نجد أن الخاصية name ترجع القيمة undefined حتى بعد إسناد قيمة إليها و ذلك بسبب أن واصفة الوضع set تحمل القيمة الإفتراضية undefined، و لو أعطيناها قيمة أخرى غير undefined أو دالة فإننا سنحصل على خطأ (و هذا ينطبق على دالة الجلب):

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                             name: {
14 ▼                                 get: function(){
15                                      return this.val;
16                                  },
17                                  set: 'Soufyane',
18                                  // enumerable: true,
19                                  // configurable: true
20                              }
21                          });
22
```

```
▶ Uncaught TypeError: Setter must be a function: Soufyane
    at Function.create (<anonymous>)
    at app.js:12
>
```

و الآن سنقوم بتعيين دالة يتم تنفيذها عند إسناد القيمة:

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                             name: {
14 ▼                                 get: function(){
15                                      return this.val;
16                                  },
17 ▼                                set: function(_val){
18                                      console.log('تم تنفيذ دالة الوضع');
19                                      console.log(arguments);
20                                  },
21                                  // enumerable: true,
22                                  // configurable: true
23                              }
24                          });
25
```

```
> obj
◂ ▸{}
> Object.getOwnPropertyDescriptor(obj, 'name');
◂ ▸{get: ƒ, set: ƒ, enumerable: false, configurable: false}
> obj.name = 'Soufyane';
  تم تنفيذ دالة الوضع
  ▸Arguments ["Soufyane", callee: ƒ, Symbol(Symbol.iterator): ƒ]
◂ "Soufyane"
> obj.name
◂ undefined
>
```

كما تلاحظ في هذا المثال، عند عملية إسناد قيمة إلى الخاصية فإن دالة الوضع تم تنفيذها، و ذلك واضح من خلال الجملة المطبوعة "تم تنفيذ دالة الوضع"، و قد تم تمرير القيمة المسندة إليها، و لنثبت ذلك إستعنّا بشبه المصفوفة، أو المتغير arguments (لو تتذكر موضوع الدوال)، و الذي يحتفظ بالوسائط المُمرّة إلى الدالة. إذن لكي يسهل لنا التعامل مع هاته القيمة و نجري عليها أية عمليات نريد، فإننا نمرِّر متغيرا كوسيط إلى دالة الوضع و الذي سيحفظ لنا تلك القيمة:

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                           name: {
14 ▼                               get: function(){
15                                     return this.val;
16                                 },
17 ▼                               set: function(_val){
18                                     console.log('تم تنفيذ دالة الوضع');
19                                     console.log('_val = ' + _val);
20                                 },
21                                 // enumerable: true,
22                                 // configurable: true
23                             }
24                         });
25
```

```
> obj.name = 'Soufyane';
  تم تنفيذ دالة الوضع
  _val = Soufyane
◂ "Soufyane"
> obj.name
◂ undefined
> |
```

كما ترى فإن الوسيط _val يحمل القيمة المسندة إلى الخاصية name. و لكن كما ترى فإنه رغم تحديد دالة الوضع إلا أن قيمة الخاصية لم تتغير بعد عملية الإسناد و لا زالت قيمتها undefined و لكي نتخطى هذا المشكل فإنه يجب علينا أن نجعل الدالة تحدد القيمة المسندة للخاصية كقيمة لمتغير خارجي و من ثم تقوم دالة الجلب بإرجاع قيمة هذا المتغير.

```
var obj = Object.create(myPrototype ,{
                            name: {
                                get: function(){
                                    return this.val;
                                },
                                set: function(_val){
                                    this.val = _val;
                                },
                                // enumerable: true,
                                // configurable: true
                            }
                        });
```

```
> obj.name
< undefined
> obj.name = 'Soufyane';
< "Soufyane"
> obj.name
< "Soufyane"
>
```

كما يظهر في المثال، جعلنا الخاصية قابلة للكتابة عبر واصفات الوصول، و لكن قد يتبادر إلى ذهنك سؤال مفاده: لماذا إستعملنا متغيرا خارجيا، أو بالأخرى خاصية أخرى في المثال لتخزين القيمة، و جلبها بدل أن نستعمل الخاصية ذاتها؟

هذا سؤال وجيه، ولكن لو فعلنا ذلك أي إستعملنا الخاصية ذاتها بدل الإستعانة بمتغير آخر، أو خاصية أخرى فإننا سنحصل على خطا مدى Uncaught RangeError. فمحاولتنا الوصول إلى الخاصية تعني تنفيذ دالة الجلب، و بما أننا جعلناها ترجع لنا قيمة الخاصية ذاتها فإن ذلك يعني محاولة وصول أخرى للخاصية معناه تنفيذ آخر لدالة الجلب و هكذا، فتتكدس الإستدعاءات في كومة التنفيذ إلى أن يتم تجاوز الحد الأقصى لكومة الإستدعاء. الأمر ذاته بالنسبة لدالة الوضع إذا تحدث تكدسات في كومة التنفيذ.

مثال:

```
var obj = Object.create(myPrototype ,{
                            name: {
                                get: function(){
                                    return this.name;
                                },
                                set: function(_val){
                                    return this.name = _val;
                                },
                                // enumerable: true,
                                // configurable: true
                            }
                        });
```

```
> obj.name
▶Uncaught RangeError: Maximum call stack size exceeded
    at Object.get (app.js:14)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
    at Object.get (app.js:15)
> obj.name = 'Soufyane';
▶Uncaught RangeError: Maximum call stack size exceeded
    at Object.set (app.js:17)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
    at Object.set (app.js:18)
>
```

بالنسبة للمتغير this كما قلنا سابقا أنه يشير إلى الكائن الذي تم إسناد قيمة للخاصية من خلاله (بالطبع نقصد هنا الخاصية التي تحمل واصفات وصول و ليس أية خاصية أخرى عادية):

```
var myPrototype = { name: 'myPrototype' };

Object.defineProperty(myPrototype, 'lastname', {
    set: function(){
        console.log('this => ' + this.name + ' :'); // => تعني يشير إلى
        console.log(this);
    }
})

var obj = Object.create(myPrototype); // obj إلى myPrototype توريث خصائص

obj.name = 'obj';
```

```
> obj
◂ ▶{name: "obj"}
> myPrototype
◂ ▶{name: "myPrototype"}
> obj.lastname = 'Hedidi'
  this => obj :
  ▶{name: "obj"}
◂ "Hedidi"
> myPrototype.lastname = null;
  this => myPrototype :
  ▶{name: "myPrototype"}
◂ null
>
```

إنتبه! فقد قلنا سابقا أنه لا يمكن الجمع بين واصفات البيانات و واصفات الوصول، و فعل ذلك ينتج خطأً:

```
var obj = Object.create(myPrototype, {
                            name: {
                                value: 'Soufyane',
                                get: function(){
                                    return 'Soufyane';
                                }
                            }
                        });
```

```
Uncaught TypeError: Invalid property descriptor. Cannot both specify accessors and a value or writable attribute, #<Object>
    at Function.create (<anonymous>)
    at app.js:3
```
app.js:3

لقد حصلنا على خطأ حين حاولنا الجمع بين واصفة بيانات و واصفة وصول. رسالة الخطأ مفادها أنّ واصفات الخاصية غير صالحة. و لا يمكن تحديد كل من واصفات الوصول و واصفة القيمة أو واصفة قابلية الكتابة.

واصفة قابلية العد (enumerable):

تعمل هذه الواصفة -في حال كانت قيمتها false- على إخفاء الخاصية من الكائن نفسه، و أيضا الحلقة for ... in، و الطرق التي تتعامل مع الكائنات. حيث أن هذه الخاصية غير مرئية، إلا أنه يمكن الوصول إليها بالطريقة العادية، أي طريقة التنقيط. يمكن إستعمال هاته الواصفة مع واصفات البيانات أو واصفات الوصول.

```
var obj = Object.create(myPrototype, {
                            name: {
                                value: 'Soufyane',
                                writable: true,
                                enumerable: true,
                                // configurable: false
                            },
                            lastname: {
                                get: function(){
                                    return 'Hedidi';
                                },
                                set: undefined,
                                enumerable: false,
                                // configurable: false
                            }
                        });

for (var key in obj){
    console.log( key + ' : ' + obj[key])
}
```

```
    name : Soufyane
    greet : function(){
            console.log('Hello ' + this.name);
        }
    printFullName : function(){
            console.log(this.name + ' ' + this.lastName);
        }
> obj.lastname
"Hedidi"
>
```

كما ترى فإن الخاصية lastname لم يتم عدها و ذلك بسبب الواصفة enumerable التي تحمل القيمة false، و بالتالي منعت الخاصية من أن يتم عدها بالرغم من أننا تمكنا من الوصول إليها. و كما قلنا في التعريف أنه عند إعطائها القيمة false (أو عدم تحديدها) فإن الخاصية لا يمكن أن تظهر أيضا بإستعمال الوظيفة keys() التابعة للدالة Object، و لا تتمكن الوظيفة assign() من نسخها إلى الكائن الجديد و أيضا لا تظهر بالنسبة لمعامل الإنتشار spread[1] و الأمر الآخر هو أنه لا يتم عمل "تسلسل" (serialized) لها عند إستخدام الوظيفة stringify() التابعة للدالة JSON.

```
> obj.lastname
"Hedidi"
> Object.keys(obj);
▶ ["name"]
> var ob = Object.assign({}, obj, {a: 13});
undefined
> ob
▶ {name: "Soufyane", a: 13}
> // إستعملنا المعامل spread لتضمين خصائص obj في obj2
  obj2 = {age: 26, country: 'Algeria', ...obj}
▶ {age: 26, country: "Algeria", name: "Soufyane"}
> JSON.stringify(obj);
"{"name":"Soufyane"}"
>
```

كما ترى في الصورة فإنه لم يتم إرجاع الخاصية lastname لأنها غير قابلة للعد و كذلك لم تتمكن الوظيفة assign() من نسخها إلى الكائن الجديد، كما أن معامل الإنتشار من الوصول إليها، و أخيرا لم تدرج في التسلسل أو السلسلة الناتجة عن الوظيفة stringify().

واصفة قابلية الضبط (configurable):

---

[1] معامل الإنتشار spread هو المعامل الذي يمرر مثلا إلى الدالة و تسبقه ثلاث نقاط، مثلا: myFunc(a, ...b) لهذا المعامل إستعمالات عديدة.

كما قلنا سابقا بأن هاته الواصفة تعمل على التحكم في قابلية ضبط واصفات الخاصية، و كذلك إمكانية حذف الخاصية بحد ذاتها، ففي حال كانت الخاصية غير قابلة للضبط أي configurable: false فإنه لا يمكن حذفها، و لا يمكن ضبط الواصفات الأخرى إلا في حالة واحدة:

إذا كانت الخاصية قابلة للكتابة أي writrable: true فيمكننا حينها أن نحولها إلى خاصية غير قابلة للكتابة أي writrable: false، و بعد ذلك أية محاولات تغيير للمواصفات أو تحديث لقيمة الخاصية لا يمكننا القيام بها، و بهذا الشكل تصبح الخاصية مجمّدة.

قبل أن نتعامل مع واصفة الضبط وجب علينا أن نشرح الوظيفة[1] defineProperty() التابعة للدالة Object و التي ستمكّننا فيما بعد من التعديل على واصفات خصائص موجودة، و حتى إضافة واحدة جديدة مع تحديد مواصفاتها:

```
11
12 ▼ var obj = Object.create(myPrototype ,{
13 ▼                             name: {
14                                       value: 'Soufyane',
15                                       writable: true,
16                                       enumerable: true,
17                                       configurable: true
18                                   }
19                               });
20
21 ▼ Object.defineProperty(obj, 'lastname', {
22                                       value: 'Hedidi',
23                                       writable: true,
24                                       enumerable: true,
25                                       configurable: false
26                                   })
27
```

```
> obj
< ▶ {name: "Soufyane", lastname: "Hedidi"}
>
```

تحديث الواصفات لأي خاصية يتم بنفس الطريقة، حيث يتم تمرير الكائن ثم الخاصية المعنية على شكل سلسلة نصية ثم التعديلات:

```
28 ▼ Object.defineProperty(obj, 'name', {
29                                       value: 'Soufyane',
30                                       writable: true,
31                                       enumerable: false,
32                                       configurable: false,
33                                   })
34
```

---

[1] الوظيفة defineProperty() تم إدخالها في es5 و تدعمها أغلب المتصفحات الجديدة.

و الآن يمكننا أن نتعامل مع الواصفة configurable لنرى تأثيرها على الخاصية:

```
> obj
< ▶ {lastname: "Hedidi", name: "Soufyane"}
> 'use strict'
  obj.name = 'Omar'; // هذه العملية صالحة لأن الخاصية قابلة للكتابة
< "Omar"
> 'use strict'
  Object.defineProperty(obj, 'name', { value: undefined }); // صالحة أيضا
< ▶ {lastname: "Hedidi", name: undefined}
> // العملية التالية غير صالحة لأن الخاصية غير قابلة للضبط
  Object.defineProperty(obj, 'name', { enumerable: true }); // سينتج خطأ
✖ ▶ Uncaught TypeError: Cannot redefine property: name
      at Function.defineProperty (<anonymous>)
      at <anonymous>:2:8
> 'use strict'
  Object.defineProperty(obj, 'name', { writable: false }); // صالحة
< ▶ {lastname: "Hedidi", name: undefined}
> Object.getOwnPropertyDescriptor(obj, 'name')
< ▶ {value: undefined, writable: false, enumerable: false, configurable: false}
> // هذه العملية غير صالحة لأن الخاصية أصبحت غير قابلة للكتابة
  Object.defineProperty(obj, 'name', { value: 26 }); // سينتج خطأ
✖ ▶ Uncaught TypeError: Cannot redefine property: name
      at Function.defineProperty (<anonymous>)
      at <anonymous>:2:8
> // هذه العملية غير صالحة لأن الخاصية غير قابلة للضبط و قد أصبحت مجمدة
  Object.defineProperty(obj, 'name', { writable: true }); // سينتج خطأ
✖ ▶ Uncaught TypeError: Cannot redefine property: name
      at Function.defineProperty (<anonymous>)
      at <anonymous>:2:8
> |
```

كما ترى في الصورة، غيرنا قيمة الخاصية name بطريقتين و قد كانتا صالحتين رغم أن الخاصية غير قابلة للضبط، و ذلك لأنها قابلة للكتابة أو التحديث (لقد تعمدت إستعمال الوضع الصارم حتى أوضح بأن العملية صالحة و لا تعطي أي خطأ).

بعدها حاولنا تغيير قابليتها للعدّ فلم نتمكن من ذلك و أعطانا هذا خطأً.

و كما أسلفنا سابقا أنه في حال كون الخاصية غير قابلة للضبط، بينما قابلة للكتابة، فإنه يمكننا تغير قابلية الكتابة إلى عدمها فقط؛ و بعد ذلك لا يمكننا أن نغيّر في الواصفات و كما ترى قد قمنا بذلك و حصلنا على خطأ، و بهذا أصبحت الخاصية مجمدة، أي لا يمكن تعديلها و لا ضبط مواصفاتها و لا حتى حذفها بإستعمال المعامل delete:

```
> delete obj.name; // لا يمكن حذف الخاصية name
< false
> obj
< ▶ {lastname: "Hedidi", name: undefined}
> |
```

و بالمناسبة هناك وظيفة تقوم بتجميد أي كائن من أن تعدّل خواصه أو تحذف تماما كما لو أننا ضبطنا مواصفاتها إلى عدم إمكانية الضبط و الكتابة أو التحديث؛ هذه الوظيفة هي freeze() التابعة للدالة Object:

```
11
12 var obj = Object.create(null ,{
13                               name: {
14                                   value: 'Soufyane',
15                                   writable: true
16                                   }
17                               });
18
19 Object.freeze(obj); // obj تم تجميد الكائن
20
```

```
> obj
◂ {lastname: "Hedidi", name: "Soufyane"}
> Object.freeze(obj);
◂ {lastname: "Hedidi", name: "Soufyane"}
> obj.age = 26; // إضافة خاصية جديدة بالطريقة الحرفية
◂ 26
> obj.name = 'Omar'; // name تحديث قيمة الخاصية
◂ "Omar"
> delete obj.lastname; // name حذف للخاصية
◂ false
> // سينتج خطأ لأن الكائن مجمّد و غير قابل للتوسعة
  Object.defineProperty(obj, 'contry', { value: 'Algeria' });
✖ ▸Uncaught TypeError: Cannot define property contry, object is not extensible
      at Function.defineProperty (<anonymous>)
      at <anonymous>:2:8
> // تم تجاهل كل التغييرات السابقة لأن الكائن مجمّد
  obj; // {name: 'Soufyane'}
◂ {lastname: "Hedidi", name: "Soufyane"}
>
```

كما ترى، فإنه قد تم تجميد الكائن obj عبر الوظيفة freeze() و لم يعد قابلا للتعديل، و لو إستعملنا الوضع الصارم لظهرت لنا رسالة خطأ عند كل محاولة تعديل من إضافة أو تحديث أو حذف. جرب إستعمال الوضع الصارم قبل كل سطر و تحقق من النتيجة.

هذا كل ما يخص الخصائص و واصفاتها. لنعد إلى النموذج المبدئي بعد أن كنا قد خصصناه إلى الكائن الذي نريد، لننشئ الآن كائنا بدون نموذج مبدئي:

```
1 var obj = Object.create(null);
2
3
```

كما ترى في هذا المثال، لقد أنشئنا كائنا عبر الوظيفة create() و مرّرنا لها القيمة null مكان النموذج المبدئي، و نعني بهذا أننا لا نريد أن يكون لهذا الكائن نموذج مبدئي، أي ببساطة لا يرث من أي كائن. شاهد النتيجة في وحدة التحكم:

```
> obj
< ▼{}
      No properties
```

يمكننا أن نضيف له الخصائص بشكل عادي إما أثناء التعريف أو بعده، كل ما في الأمر أن هذا الكائن لا يملك نموذجا مبدئيا يرث منه.

و لكن إنتبه إلى الكائنات التي لا تملك نموذج مبدئي أنه لا يمكنك إستعمال الوظائف التابعة للكائن Object و محاولتك لفعل ذلك سينتج خطأ، تفحص هذا المثال:

```
1  var obj1 = Object.create({});
2  var obj2 = Object.create(null);
3
```

```
> "obj1 : " + obj1;
< "obj1 : [object Object]"
> "obj2 : " + obj2; // سينتج خطأ
✖ ▶Uncaught TypeError: Cannot convert object to primitive value
      at <anonymous>:1:11
> |
```

لاحظ عدم إمكانية جمع كائن بدون نموذج مبدئي مع سلسلة، و ذلك يرجع لأن هذه الكائن لا يملك الوظيفة toString().

```
> alert(obj1); // يعمل بشكل عادي
< undefined
> alert(obj2); // سينتج خطأ
✖ ▶Uncaught TypeError: Cannot convert object to primitive value
      at <anonymous>:1:1
> |
```

```
> alert(obj1); // يظهر [object Object] في نافذة تنبيه
< undefined
> alert(obj2); // سينتج خطأ
✖ ▶Uncaught TypeError: Cannot convert object to primitive value
      at <anonymous>:1:1
> |
```

```
> obj1.toString();
"[object Object]"
> obj2.toString();
Uncaught TypeError: obj2.toString is not a function
    at <anonymous>:1:6
>
```

```
> obj1.constructor
ƒ Object() { [native code] }
> obj2.constructor
undefined
>
```

لنشاهد جانب آخر

إنشاء الكائنات بإستخدام المعامل new مع الدوال:

يحتوي هذا الفصل أمورا هامة جدا، تستعمل بكثرة في المشاريع و يتم شرحها في أغلب المواقع و الكتب العربية لكن دون تفصيل لما يحدث في الخفاء، بفهمك لهذه الأمور سيفتح لك لغز كبير يقبع وراء الشيفرات المصدرية لأشهر إطارات العمل و المكتبات مثل JQuery، Underscore ... و العديد، بل ستتمكن من إنشاء مكتبتك\إطار العمل الخاص بك بكل مرونة و متعة.

لو تتذكر فصل المعاملات و التي قلنا عنها أنها دوال لكنها خاصة إذ تختلف عن الدوال العادية شكلا، و بالطبع هذه المعاملات قد تأخذ وسيط أو إثنان أو ثلاثة. الآن سنتعامل مع المعامل الأحادي new و الذي يأخذ وسيطا واحدا حيث يكون عبارة عن دالة، تدعى هاته الدالة في هذه الحالة بالدالة البانية و ذلك لأنها تبني لنا الكائن الجديد المنشئ و تعده لنا.

لنأخذ مثالا

```
1  function Person(){}
2
3  var obj = new Person(); // إنشاء كائن
4
```

```
> obj
▶ Person {}
>
```

و الآن نأتي لشرح ما حدث في الخلفية:

عند تعريفنا للدالة Person فإنها تعتبر دالة عادية، و المتغير this الخاص بها يشير في هذه الحالة إلى الكائن العام Window، و لكن عندما نمرر هاته الدالة إلى المعامل new تحدث أشياء خاصة كالتالي:

أولا، يتم إنشاء كائن جديد فارغ {}.

ثانيا، يتم إستدعاء أو تنفيذ الدالة Person()، و كما نعلم سابقا بأنه عند تنفيذ أي دالة، فإنه يتم توليد متغير يدعى this، و يشير هذا المتغير إلى كائن ما حسب حالة إستدعاء الدالة. في هذه الحالة عند إستعمالنا المعامل new، فإن المحرك يغيرّ ما يشير إليه المتغير this و يجعله يشير إلى ذلك الكائن الجديد الفارغ.

ثالثا، يتم تحديد النموذج المبدئي لهذا الكائن ليكون الخاصية prototype التابعة للدالة البانية (الدالة Person() في هذا المثال). (معلوم أن الدوال عبارة عن كائنات و يمكن أن تكون لها خصائص).

رابعا في حال أن الدالة لا ترجع أي قيمة فإنه يتم إرجاع ذلك الكائن الجديد بعد إجراء العمليات التي تحملها الدالة و إلا فإنه يتم إرجاع القيمة المرجعة من الدالة.

إذن كل ما في الأمر أن الدالة تُستدعى بشكل عادي عن طريق المعامل new، فقط يتم تغيير الإشارة المرجعية للمتغير this ليشير إلى الكائن الجديد، و بهذا نستطيع تحديد الخصائص التي ستضاف إلى الكائن الفارغ عن طريق هاته الدالة. أيضا يُضبط النموذج المبدئي للكائن الجديد، و في الأخير يتم إرجاعه في حال لم ترجع الدالة أي شيئ.

فلنأخذ أمثلة لتوضيح ما سبق:

```
1  var savedThis;
2  function Person(){
3      console.log('تم تنفيذ الدالة');
4      console.log(this);
5      savedThis = this;
6  }
7
8  var obj = new Person; // إنشاء كائن
9
10 console.log(savedThis === obj); // true
11
```

```
تم تنفيذ الدالة                    app.js:3
▶ Person {}                        app.js:4
true                               app.js:10
>
```

أولا، عرّفنا متغيرا لنخزن فيه ما يشير إليه المتغير this الذي يتولى محرك جافا سكريبت تعيين إشارته المرجعية أثناء بناء الكائن الجديد.

بعد ذلك عرفنا دالة البناء و التي تقوم بطباعة جملة لنتأكد من أن الدالة تم تنفيذها عند إنشاء الكائن، و أخيرا أنشئنا الكائن obj بإستعمال المعامل new مع الدالة Person، و أنظر أنها كتبت بدون قوسين و هذا أمر ممكن، فقط القوسين لتمرير وسائط للدالة إن كان هناك وسائط تستقبلها الدالة.

و كما تشاهد النتيجة فإن الدالة قد تم تنفيذها و طبعت لنا الجملة "تم تنفيذ الدالة"، و بعدها طبعت لنا ما يشير إليه المتغير this أثناء ذلك الإستدعاء و هو الكائن الجديد و إسم الدالة البانية الذي قبله يعني أن الكائن من نوع Person، و من هنا تأتي فلسفة الأصناف و الحالات (instances) في الجافا سكريبت، حيث أنها تختلف عن باقي اللغات في إنشاء الأصناف، إذ بدل أن تستعمل الكلمة المفتاحية class، فإنها تعتمد على الدوال في تحديد خصائص و طرق الصنف. و إنشاء الحالات (أو النسخ أو المثيلات) من هذا الصنف يكون بإستعمال المعامل new المقتبس من لغة جافا لأجل جذب مبرمجيها حيث كانت لغة جافا ذات شهرة واسعة عند إطلاق لغة جافا سكريبت.

و الآن سنعدّل في الدالة البانية لنحدد خصائص و طرق لتضاف إلى الكائن الجديد الذي سننشئه:

```
function Person(name, lastname, age){
    console.log(this);
    this.name = name || 'anonymous';
    this.lastname = lastname || 'anonymous';
    this.age = age || 'alive!';
    this.bio = function(){
        console.log("I'm " + this.lastname +
                    " " +this.name +
                    ", I'm " + this.age);
    }
}

var souf = new Person('Soufyane', 'Hedidi', 26); // إنشاء نسخة\مثيل\حالة
console.log(souf);
console.log(souf.bio());

var anonym = new Person(); // إنشاء نسخة\مثيل\حالة
console.log(anonym);
console.log(anonym.bio());
```

```
▶ Person {}                                                    app.js:2
▶ Person {name: "Soufyane", lastname: "Hedidi", age: 26, bio: ƒ}   app.js:14
I'm Hedidi Soufyane, I'm 26                                    app.js:7
undefined                                                      app.js:15
▶ Person {}                                                    app.js:2
▶ Person {name: "anonymous", lastname: "anonymous", age: "alive!", bio: ƒ}  app.js:18
I'm anonymous anonymous, I'm alive!                            app.js:7
undefined                                                      app.js:19
>
```

كما ترى في هذا المثال أنشأنا مثيلين من الصنف Person، و يملك كل منهما ثلاث خصائص و وظيفة واحدة تقوم بطباعة جملة في وحدة التحكم، و أثناء إنشائهما تكفّل المحرك بتحديد ما يشير إليه المتغير this الخاص بالدالة البانية ليشير في كل مرة إلى الكائن الفارغ الجديد الذي يتم بنائه، و بهذا تم إضافة الخصائص الثلاث و الطريقة إلى كل كائن.

إذن الدالة البانية هي دالة عادية تستعمل لبناء الكائنات، و عند تمريرها إلى المعامل new فإن المتغير this -الذي يتم إنشائه خلال مرحلة الإنشاء في سياق التنفيذ- يشير إلى كائن جديد فارغ، و هذا الكائن يتم إرجاعه من الدالة بشكل تلقائي عند إنتهاء تنفيذ الدالة.

إنتبه في حال كانت الدالة البانية ترجع كائنا، فإن هذا الكائن هو الذي يأخذ بعين الإعتبار:

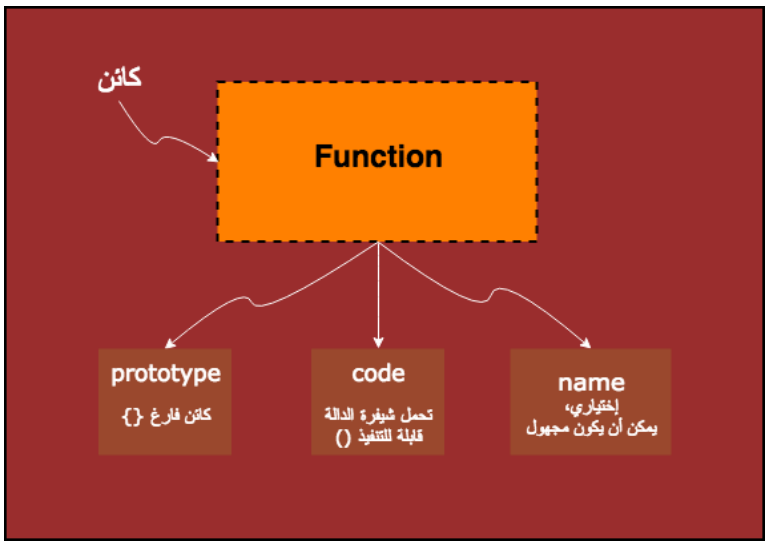
```
function Person(name, lastname, age){
    console.log(this);
    this.name = name || 'anonymous';
    this.lastname = lastname || 'anonymous';
    this.age = age || 'alive!';
    this.bio = function(){
        console.log("I'm " + this.lastname +
                    " " + this.name +
                    ", I'm " + this.age);
    }
    return {name: 'username'};
}

var souf = new Person('Soufyane', 'Hedidi', 26); // إنشاء نسخة\مثيل\حالة
console.log(souf);
```

```
▶ Person {}                    app.js:2
▶ {name: "username"}           app.js:15
> |
```

كما ترى في هذا المثال فإن الدالة البانية تُرجع كائنا يحمل خاصية واحدة و هو ما تم إرجاعه بدل الكائن الجديد.

و الآن لننتقل إلى النموذج المبدئي للكائنات المبنية عن طريق الدوال البانية:

هل تتذكر موضوع الدوال الذي قلنا فيه أن كل دالة عبارة عن نوع خاص من الكائنات؟ و لها بعض الخصائص كالخاصية name التي تحمل إسم الدالة و الخاصية code التي تحمل شيفرة الدالة؟ هناك خاصية إضافية تدعى prototype أي النموذج المبدئي، و تحمل هاته الخاصية كائنا فارغا، هذه الخاصية غير مخفية و يمكنك إستعمالها عند إستعمال الدالة كدالة بانية.



إنتبه إلى أن الخاصية prototype لا تعني النموذج المبدئي للدالة بحد ذاتها! و إنما ستصبح النموذج المبدئي لأي كائن أُنشئ عبر الدالة بإستخدام المعامل new، يعني أن جميع الكائنات المنشأة عبر دالة بانية ترث من الخاصية prototype و التي بدورها تعَد كائنا.

و بالمثال يتضح المقال:

```
> souf.__proto__
◂ ▾{constructor: ƒ}
    ▸constructor: ƒ Person(name, lastname, age)
    ▸__proto__: Object
> Person.prototype
◂ ▾{constructor: ƒ}
    ▸constructor: ƒ Person(name, lastname, age)
    ▸__proto__: Object
> Person.prototype === souf.__proto__
◂ true
>
```

من خلال المثال السابق نجد أن النموذج المبدئي للكائن souf هو كائن يحمل خاصية تشير إلى الدالة البانية Person و قد وصلنا إليه عن طريق الخاصية __proto__ التي توفرها وحدة التحكم في متصفح كروم.

و عند الوصول إلى الخاصية prototype على كائن الدالة Person نجد أن هذه الخاصية تشير إلى نفس الكائن السابق و هو النموذج المبدئي للكائن souf و عملية المقارنة تثبت ذلك.

و سبب كون الخاصية prototype أصبحت نموذجا مبدئيا للكائن souf هو إستعمال الدالة Person كدالة بانية مع المعامل new، و بالطبع أي كائن آخر يتم إنشائه من هذه الدالة فإن نموذجه المبدئي هو الخاصية prototype.

```
> var obj = new Person('human');
  ▼Person {} i
      age: "alive!"
    ▶bio: ƒ ()
      lastname: "anonymous"
      name: "human"
    ▼__proto__:
      ▶constructor: ƒ Person(name, lastname, age)
      ▶__proto__: Object
undefined
> obj.__proto__ === Person.prototype
true
>
```

و الآن لنضف بعض الخصائص و الوظائف إلى الكائن prototype:

```
11
12  Person.prototype.lang = 'ar';
13
14  Person.prototype.getName = function(){
15      return this.name;
16  };
17
18  var souf = new Person('Soufyane', 'Hedidi', 26); // إنشاء مثيل
19  console.log(souf.getName());
20  console.log(souf.lang);
21
```

```
Soufyane                    app.js:19
ar                          app.js:20
>
```

كما ترى، بإضافتنا للخاصية lang و الوظيفة getName() تمكّن الكائن من الوصول إليهما عبر البحث في سلسلة النماذج المبدئية و قد عثر عليهما في الكائن prototype التابع للدالة البانية Person و أي كائن آخر يتم إنشائه عبر هاته الدالة البانية يستطيع الوصول إلى هاته الخصائص و الوظائف:

```
21
22  var reda = new Person('Reda');
23  console.log(reda.getName());
24  console.log(reda.lang);
25
```

```
Reda                        app.js:23
ar                          app.js:24
>
```

و فائدة هذه الخاصية (أي prototype) هو أنه في حالة أردنا إضافة خصائص و طرق جديدة لجميع الكائنات بعد إنشائها عن طريق الدالة البانية، فإننا نضيفها لهذه الخاصية أو بالأحرى هذا الكائن.

أما الميزة القوية فهي كالتالي:

تخيل أنك تريد إنشاء مئة كائن بإستعمال الدالة البانية، و كما تعرف أن الدالة البانية تضيف الخصائص و الوظائف المحددة داخلها إلى الكائن الجديد المُنشأ. و بالنسبة لأي كائن فإن لكل خاصية من خصائصه مساحة في الذاكرة.

إذن بإنشائنا لمئة كائن من خلال الدالة البانية يعني مئة قدر من المساحة محجوزة في الذاكرة، و لكل كائن خصائص و وظائف، مما يعني إستهلاك المزيد من مساحة الذاكرة لتخزين تلك الخصائص و الوظائف، و هذه تعد سلبية لما لها تأثر على أداء التطبيق أو الصفحة.

هنا يأتي دور الخاصية prototype و التي تصبح نموذجا مبدئيا للمئة كائن المنشئة عبر الدالة البانية، أي أنها جميعا تتشارك نفس النموذج المبدئي، و بهذا فإنه بإضافة أي خاصية أو وظيفة لهذا النموذج المبدئي تصبح متوفرة لجميع تلك الكائنات و منه نكون قد تجنبنا تكرار نفس الخصائص و الوظائف لكل كائن، أي تم حل مشكلة إهدار مساحة الذاكرة.

و كمعلومة أخيرة، قد تتشارك الكائنات من نفس النوع الوظائف إلا أنها تختلف في الخصائص. فمثلا جميع السيارات تتشارك نفس الوظائف منها السير التوقف الدوران، إلا أنها تختلف من حيث الألوان و الشكل و الوزن ... إلخ. و إستنادا على هذا في برمجة الكائنات فإنه يفضل تعريف الخصائص داخل الدالة البانية حيث سيتم إعطاء كل كائن خصائصه الخاصة به، أما بالنسبة للوظائف فيفضل إضافتها عن طريق النموذج المبدئي، أي خارج الدالة البانية، و ذلك تجنبا لهدر مساحة الذاكرة، و بسبب أن الكائنات تتشارك نفس الوظائف. اللهم إذا أردت شيئا مختلفا فهنا لكل مقام مقال.

هذه المنهجية معمول بها في أغلب أطر العمل و المكتبات و هذا يعد من أفضل الممارسات.

عند تعريفنا للدوال البانية كنا قد إبتدئنا إسمها بحرف كبير، لأن هذا تقليد شائع بين مبرمجي لغة جافا سكريت و ليس أمرا إجباريا في اللغة، فقط للتمييز بين الدوال العادية و الدوال البانية. حتى الدوال المبنية في اللغة تتبع هذا النهج إذ تجدها تبتدأ جميعا بحرف كبير و بقية الحروف صغيرة:

```
> Object
ƒ Object() { [native code] }
> Array
ƒ Array() { [native code] }
> Function
ƒ Function() { [native code] }
> Boolean
ƒ Boolean() { [native code] }
> Number
ƒ Number() { [native code] }
> Symbol
ƒ Symbol() { [native code] }
>
```

هذا التقليد أو النهج له فائدة في مساعدتنا على تنقيح الأخطاء التي قد نرتكبها، فأحيانا ننسى أن نكتب المعامل new قبل الدوال البانية، و عند البحث عن المشكل سنجد أن تلك الدوال تبتدأ بحرف كبير مما يوحي بأنها موجهة لبناء الكائنات، إذن من المحتمل أننا نسينا سبْقها بالمعامل new:

```
18  var souf = Person('Soufyane', 'Hedidi', 26); // إنشاء متيل
19  console.log(souf.getName());
20  console.log(souf.lang);
```

```
▶ Uncaught TypeError: Cannot read property 'getName' of undefined
    at app.js:19                                    app.js:19
>
```

كما ترى، حصلنا على خطأ عند إستدعاء الوظيفة getName()، و السبب راجع إلى أننا إستدعينا الدالة Person فقط! و هي لا ترجع قيمة صريحة، إذن سيرجع محرك جافا سكريبت القيمة undefined كقيمة إفتراضية، و محاولة إستدعاء دالة على undefined غير ممكن و ينتج خطأً.

في الحقيقة محرك جافا سكريبت لا يعرف بأنك تنوي إنشاء كائن من دالة ما، و لن يعرف متى يجب أو لا يجب عليك أن تستعمل المعامل new، إذن يمكنك الوقوع في أخطاء غير مقصودة و التي لن يلفت المحرك نظرك إليها أثناء تنفيذ البرنامج، لينتهي برنامجك بأخطاء غير متوقعة.

و لهذا يستحسن إبتداء الدوال البانية بأحرف كبيرة من أجل تمييزها، و يعتبر تقليدا متعارف عليه.

الآن و بعد فهمنا للدوال البانية، دعنا نتكلم قليلا عن الدوال البانية المبنية في اللغة:

هذه الدوال تأتي ضمن تركيبة اللغة، و هي جاهزة للإستعمال داخل محرك جافا سكريبت، فكيف تتعامل معها؟

لنأخذ بعض الأمثلة:

```
> var s = new String("Soufyane");
< undefined
> s
< ▼String {"Soufyane"}
    0: "S"
    1: "o"
    2: "u"
    3: "f"
    4: "y"
    5: "a"
    6: "n"
    7: "e"
    length: 8
  ▶ __proto__: String
    [[PrimitiveValue]]: "Soufyane"
>
```

قمنا في هذا المثال بإستدعاء الدالة البانية String المبنية في اللغة عبر المعامل new. و النتيجة عبارة عن كائن و ليس سلسلة نصية، هذا الكائن يحمل العديد من الخصائص، و الخاصية المظللة بالأصفر تدعى بالمغلفة، و تحمل القيمة البدائية أو البسيطة.

بما أن s عبارة عن كائن هذا يعني أن له نموذجا مبدئيا ألا وهو الخاصية prototype التابعة للكائن String و منه فأنه يستطيع الوصول إلى ما تحتويه هذه الخاصية من وظائف و خصائص. أولا دعنا نرى على ماذا تحتوي:

```
> String.prototype
< ▼String {"", constructor: ƒ, anchor: ƒ, big: ƒ, blink: ƒ, …}
  ▶ anchor: ƒ anchor()
  ▶ big: ƒ big()
  ▶ blink: ƒ blink()
  ▶ bold: ƒ bold()
  ▶ charAt: ƒ charAt()
  ▶ charCodeAt: ƒ charCodeAt()
  ▶ codePointAt: ƒ codePointAt()
  ▶ concat: ƒ concat()
  ▶ constructor: ƒ String()
  ▶ endsWith: ƒ endsWith()
  ▶ fixed: ƒ fixed()
```

إقتبست لك شيئا مما يحتويه الكائن prototype و لك أن تطلع على البقية. و كما ترى فإنه يحتوي على الخاصية length و مجموعة كبيرة من الوظائف الشهيرة و التي لابد و أنك تعاملت معها مسبقا مرارا و تكرارا، مثل concat(), indexOf(), match(), slice(), toUpperCase() ... إلخ.

إذن يمكن للكائن s أن يصل إلى هاته الطرق عبر سلسلة النموذج المبدئي:

```
> s.toUpperCase();
< "SOUFYANE"
> s.indexOf('a');
< 5
> |
```

حسننا، قد تقول أنه يمكن إستعمال هاته حتى مع القيم البدائية للسلاسل النصية، نعم هذا صحيح و لكن هذا لا يعني أن القيم البدائية تمتلك خصائص و وظائف، فهي عبارة عن قيم صرفة مجردة و ليست كائنات. أما فيما يتعلق بإمكانية إستعمال الوظائف مع القيم البدائية بطريقة التنقيط، فهذا يرجع إلى أن محرك جافا سكريبت يغلف تلك القيمة البدائية داخل كائن String كي يستطيع الوصول إلى الوظيفة المعنية من خلال سلسلة النموذج المبدئي، ثم يرجع لك النتيجة المطلوبة.

الأمر يبدو كما في المثال أعلاه إلا أنه يحدث في الخلفية تلقائيا من قبل المحرك.

هذه العملية تنطبق على بقية الأنواع البدائية للقيم حيث يتم تحويل القيمة البدائية إلى كائن يقابلها في النوع ثم تجرى العملية المطلوبة.

حسنا، كما نعلم أن الدوال لها الخاصية prototype و التي تعمل عند إستعمال الدالة للبناء، و بالطبع إضافة الخصائص و الوظائف لهاته الخاصية يصبح متاحا لجميع الكائنات المنشأة عبر الدالة، فهل هذا يعني أنه يمكن إضافة وظائف و خصائص جديدة إلى خاصية prototype التابعة لدالة مبنية في اللغة؟ نعم هذا ممكن جدا، و هذا ما يجعل لغة جافا سكريبت ديناميكية؛ فأنت يمكنك تسويغها كيفما تريد، لنأخذ مثالا يوضح هذا:

```
String.prototype.invertCase = function(){
    var tempVar = this.toString();
    var result = '';
    for(var i = 0; i < tempVar.length; i++){
        if( tempVar[i] === tempVar[i].toLowerCase() ){
            result += tempVar[i].toUpperCase();
        }
        else{
            result += tempVar[i].toLowerCase();
        }
    }
    return result;

}
```

```
> "sOUfYANE".invertCase()
< "SouFyane"
> |
```

كما ترى في هذا المثال لقد أضفنا ميزة جديدة إلى اللغة، ألا و هي عكس حالة أحرف أيّ سلسلة نصية، و بالطبع تم هذا عن طريق إضافة الوظيفة invertCase إلى الخاصية prototype التابعة للدالة البانية String.

كل ما تقوم به هاته الوظيفة هو كالآتي:

- تخزين القيمة البدائية التي تم تغليفها داخل الكائن الذي يشر إليه المتغير this عن طريق تحويلها إلى سلسلة نصية (السطر 2).
- الدخول في حلقة تمر على جميع أحرف السلسلة النصية و التحقق من حالة الحرف.
- إن كان الحرف صغيرا تجعله كبيرا و العكس صحيح. ثم تضيفه إلى المتغير result الذي يحمل النتيجة النهائية الذي يتم إرجاعه في الأخير.

مثال آخر:

```
1 ▼ String.prototype.strInObj = function(){
2       return this;
3   }
```

```
> var txt = 'text';
< undefined
> var txtObj = txt.strInObj();
< undefined
> txt
< "text"
> txtObj
< ▶ String {"text"}
> txt === txtObj
< false
>
```

في هذا المثال أردت أن أوضح كيف يقوم محرك جافا سكريبت بإنشاء كائنات من القيم البدائية أو بالأحرى تغليفها داخل كائنات، ليتسنى له الوصول إلى الوظائف أو الخصائص المطلوبة عن طريق البحث في سلسلة النموذج المبدئي.

هناك بعض الجوانب المربكة بالنسبة للدوال البانية المبنية في اللغة حيث ستصادف أخطاء غير متوقعة إن لم تكن على دراية بما يحدث:

```
Number.prototype.isPair = function(){
    return this % 2 === 0;
}
```

```
> 3.isPair()
Uncaught SyntaxError: Invalid or unexpected token      VM45:1
>
```

شاهد الخطأ الذي حصلنا عليه حيث أن هذا الخطأ تركيبي أي يتعلق ببناء الجملة، مما يعني أن معاملتنا للقيمة البدائية 3 كالكائن غير ممكن.

صحيح أن محرك جافا سكريبت قويّ بما فيه الكفاية ليحوّل سلسلة نصية إلى كائن بشكل تلقائي من أجل إتمام العملية المطلوبة، لكنه لن يحول رقما إلى كائن تلقائيا.

الطريقة isPair تعمل على الكائنات المنشأة عبر الدالة البانية Number فقط:

```
> var age = new Number(26);
undefined
> age.isPair()
true
>
```

مثال آخر:

```
> var a = 5;
undefined
> var b = new Number(5);
undefined
> a == b;
true
> a === b;
false
>
```

شاهد إختلاف النتائج عند المقارنة بين القيمة البدائية 5 التي يحملها المتغير a و الكائن 5 الذي يشير إليه المتغير b، حيث أن معامل المساواة الثنائي == يقوم بعملية التحويل القسري في حال إختلف نوع الطرفين و بعد ذلك يقارن بينهما، أما معامل المساواة الثلاثي === فيتحقق من تساوي نوع الطرفين أولا فإن كان مختلفا يرجع false مباشرة. ولهذا ينصح بإستعمال معامل المساواة الثلاثي من أجل المقارنة حتى يجنبنا أي تحويل قسري له تأثير سلبي على النتيجة.

أمر آخر يتعلق بالمصفوفات لابد أن تحتاط منه أثناء تعاملك معها:

أولا، المصفوفات في الجافا سكريبت تختلف قليلا عن المصفوفات في باقي اللغات، فهي عبارة عن كائنات، و كل عناصرها عبارة عن خصائص، أما الفهارس فهي أسماء أو لنقل المفاتيح لتلك العناصر، لنأخذ مثالا يوضح هذا الأمر:

```
1  Array.prototype.name = 'Soufyane';
2
3  var arr = ['Ahmed', 'Omar', 'Salim'];
4
5  for(prop in arr){
6      console.log(prop + ' : ' + arr[prop]);
7  }
8
```

```
0 : Ahmed                    app.js:6
1 : Omar                     app.js:6
2 : Salim                    app.js:6
name : Soufyane              app.js:6
> |
```

المتوقع من نتيجة هذا المثال هو أن يتم طباعة ثلاث سلاسل نصية في وحدة التحكم، إلا أنه تم طباعة أربعة! و هذا راجع إلى أن الحلقة for in تقوم بالبحث في الكائن بحد ذاته و في سلسلة النموذج المبدئي، و لهذا طبعت لنا الخاصية الرابعة name المعرّفة في الخاصية prototype التي تعد نموذجا مبدئيا لأي مصفوفة يتم إنشائها. و لعلمك عندما تُنشئ مصفوفة بالطريقة الحرفية [] فإنها في الأساس تستدعي new Array، فالطريقة الحرفية مجرد إختصار فقط، و الأمر سيان بالنسبة لإنشاء كائن بالطريقة الحرفية {}.

إذن في حالة المصفوفات من المستحسن إستعمال الحلقة القياسية for(...; ...; ...) من أجل الدوران على العناصر، فهذا أأمن، و ذلك لتجنب البحث في سلسلة النموذج المبدئي، أو إستعمال الطريقة hasOwnProperty() التي إستعملناها سابقا.

الكلمة المفتاحية class:

في الإصدار القادم من جافا سكريبت ES6 سيتم إدخال الكلمة المفتاحية class كطريقة أخرى لإنشاء الكائنات و ضبط النموذج المبدئي. هذه الكلمة المفتاحية تعتبر تجميلا لغويا فقط، و التجميل اللغوي هو كتابة شيئ ما بطريقة أخرى في الشيفرة، و التي لا تغيّر كيفية عمله في الخلفية. فكما رأينا سابقا أن الدوال البانية و Object.create تقومان بنفس الشيء و هو إنشاء الكائنات و ضبط نماذجها المبدئية، فإن class تقوم بنفس الشيء تماما، إذ أنها لا تغيرّ أي شيء حول كيفية قيام محرك جافا بضبط الأمور، و كيف تعمل الكائنات و النماذج المبدئية.

```
class Person {
    constructor(name, lastname){
        this.name = name || 'anonymous';
        this.lastname = lastname || 'anonymous';
        this.age = 'alive!';
    }

    getName(){
        return this.name;
    }
}

var souf = new Person('Soufyane', 'Hedidi');
console.log(souf);
```

```
▶ Person {name: "Soufyane", lastname: "Hedidi", age: "alive!"}   app.js:14
> souf.age
"alive!"
>
```

لاحظ كيف أنشئنا الصنف Person بإستخدام الكلمة المفتاحية class، فيما يخص الدالة constructor فإسمها يدل عليها، و هي دالة خاصة تعمل على تهيئة الكائن المنشأ عن طريق class، تماما مثل الدوال البانية التي رأيناها من قبل، و يجب أن يحتوي الصنف على واحدة منها فقط، و إلا فسنحصل على خطأ في حال وجد أكثر من واحدة.

أما الطريقة getName فقد تم تعريفها مباشرة بالإسم دون الحاجة إلى الكلمة المفتاحية function، حيث سيتم إضافتها إلى النموذج المبدئي. و بالنسبة لأنشاء الكائنات فإننا مازلنا بحاجة إلى إستعمال المعامل new.

هذه الطريقة لإنشاء الكائنات جميلة إلى حد ما فهي تجعل الأمر واضحا بأن هذا صنف مما يوجب إستخدام المعامل new حيث سيمنعك المحرك من الوقوع في أخطاء سخيفة كالتي تكلمنا عنها في السابق.

هناك مشكلة تصادف القادم من لغات البرمجة الأخرى كالجافا و السي++ ...إلخ عند رؤيته للكلمة المفتاحية class سيقول هذا رائع! إنها مشابهة تماما لما في اللغات السابقة، و يبدأ مباشرة في تصميم تركيبة الكائنات كما كان يفعل في اللغات السابقة. لكن في الحقيقة أن الأصناف في جافا سكريبت في حد ذاتها كائنات، و يتم إنشاء الكائنات منها، بينما في اللغات الأخرى الأصناف ليست كائنات، هي مجرد تعاريف أو عبارة عن قوالب تحدد شكل الكائنات.

لكن رغم إدخال جافا سكريبت للكلمة المفتاحية class إلا أنها لا تملك الأصناف بنفس المفهوم الذي في لغات البرمجة الأخرى.

بالمناسبة، لضبط النموذج المبدئي يتم إستعمال الكلمة المفتاحية extends كالتالي:

```
class FormalName extends Person {
    constructor(name, lastname){
        super(name, lastname);
    }

    getName(){
        return 'Mr. ' + this.name;
    }

    getFullName(){
        return this.name + ' ' + this.lastname;
    }
}
```

في هذا المثال ضبطنا النموذج المبدئي لأي كائن يتم إنشاؤه من الصنف FormalName. أيضا الدالة البانية في هذا الصنف ستستدعي الدالة البانية الخاصة بالصنف Person عن طريق الكلمة المفتاحية super و المستدعاة كما هو موضح في المثال.

لاحظ أيضا أننا تجازونا الوظيفة getName الموجودة في الصنف Person من خلال تعريفها مرة أخرى في الصنف FormalName.

```
> var fname = new FormalName('Soufyane', 'Hedidi');
< undefined
> fname.getName()
< "Mr. Soufyane"
> fname.getFullName()
< "Soufyane Hedidi"
> fname.age
< "alive!"
> fname
< FormalName {name: "Soufyane", lastname: "Hedidi", age: "alive!"}
    age: "alive!"
    lastname: "Hedidi"
    name: "Soufyane"
  ▼ __proto__: Person
    ▶ constructor: class FormalName
    ▶ getFullName: ƒ getFullName()
    ▶ getName: ƒ getName()
    ▶ __proto__: Object
>
```

نظرة على شيفرة jQuery

حسنا، إلى حد الآن أخذنا المبادئ و المفاهيم الأساسية، و التي ستفتح لنا أوسع أبواب الإحتراف في لغة جافا سكريبت، لنتمكن حينها من إنشاء مكتبات أو أطر عمل خاصة بنا، تماما مثل المكتبات و أطر العمل الشهيرة كالـ jQuery و AngularJs. دعنا إذاً نلقي نظرة على الشيفرة المصدرية لمكتبة JQuery لنرى ماهي الطريقة المعمول بها، و أيضا لنتعلم منها، فهذا يعد من أفضل الطرق لتحسين المهارات، كي نستعمل ما تعلمناه سابقا لنرى إن كنا نستطيع فهم شيئ من تركيبة و أسلوب هذ المكتبة:

```
/*!
 * jQuery JavaScript Library v3.3.1
 * https://jquery.com/
 *
 * Includes Sizzle.js
 * https://sizzlejs.com/
 *
 * Copyright JS Foundation and other contributors
 * Released under the MIT license
 * https://jquery.org/license
 *
 * Date: 2018-01-20T17:24Z
 */
( function( global, factory ) {

    "use strict";

    if ( typeof module === "object" && typeof module.exports === "object" ) { ··· } else
    { ··· }

// Pass this if window is not defined yet
} )( typeof window !== "undefined" ? window : this, function( window, noGlobal ) { ··· }
);
10365
```

سأستعمل في الشرح الإصدار الأخير أثناء كتابة هذا الكتاب ألا وهو 3.3.1، لقد قمت بطي الشيفرة لكي نرى الشكل العام للمكتبة، أولا إستهل مبرمجوا المكتبة بتعليقات جميلة لتوضيح بعض المعلومات حولها.

و كما ترى أنهم يستعملون دالة مجهولة داخل معامل التجميع كحيلة لخداع المحرك، و قد إستعملوا الطريقة الثانية التي تكلمنا عنها في فصل إنشاء دالة على الطاير، أي تنفيذ الدالة بعد إرجاعها.

هذه الدالة تستقبل وسيطين global و factory، و تستعمل بداخلها الوضع الصارم. كل ما تقوم به هو عملية تحقق من البيئة التي تعيش فيها المكتبة، و من ثم إستدعاء الوسيط الثاني حسب الشرط المحقق، الوسيط الثاني سيكون عبارة عن دالة:

```
18 ▼    if ( typeof module === "object" && typeof module.exports === "object" ) {
19
20          // For CommonJS and CommonJS-like environments where a proper `window`
21          // is present, execute the factory and get jQuery.
22          // For environments that do not have a `window` with a `document`
23          // (such as Node.js), expose a factory as module.exports.
24          // This accentuates the need for the creation of a real `window`.
25          // e.g. var jQuery = require("jquery")(window);
26          // See ticket #14549 for more info.
27          module.exports = global.document ?
28              factory( global, true ) :
29 ▼            function( w ) {
30 ▼                if ( !w.document ) {
31                      throw new Error( "jQuery requires a window with a document" );
32                  }
33                  return factory( w );
34              };
35 ▼    } else {
36          factory( global );
37      }
```

الدالة المجهولة في جميع الأحوال ستضيف هيكل المكتبة إلى الكائن العام عن طريق الدالة factory.

يمكنك تخطي الفقرات التالية إذا فهمت عمليات التحقق، و في حال تعسّر عليك ذلك، تابع الشرح المفصل:

الدالة المجهولة تتحقق أولا من أن نوع كل من module و module.exports عبارة عن كائن، هذان الكائنان موجودان في البيئات الأخرى كالـ Node.js، كما هو موضح في التعليقات.

إذا كانت الدالة تتعامل مع بيئة أخرى كالـ Node.js فإنها ستعيد إسناد قيمة الخاصية exports، و هذا عن طريق المعامل الثلاثي الذي سيتحقق من صحة الوسيط الأول (الشرط) global.document و الذي يعني وجود العقدة document ضمن الكائن العام لتلك البيئة، و هذا منطقي بما أن المكتبة موجهة للتعامل مع شجرة الـ DOM.

إن كان الشرط صحيحاً فإن exports سيشير إلى القيمة المرجعة من الدالة factory ممرا إليها الوسيط global (الكائن العام).

المثال التالي سأستعمله فقط للتوضيح:

```
<script type="text/javascript" >
    var module = {};
    module.exports = {}
</script>
<script src="../frameworks/jquery-3.3.1.js" ></script>
```

أنشأت ملف HTML و بداخله أدرجت هذه الشيفرة والتي تعرف الكائن module ثم تضيف إليه الخاصية exports و التي هي عبارة عن كائن أيضا، لا أنه تم إدراج هذه الشيفرة قبل سطر تضمين المكتبة. أردت بهذه العملية أن أوضح ما الذي تقوم به الدالة المجهولة في كان الشرط الأول صحيحا:

```
> module
{exports: ƒ}
    exports: ƒ ( selector, context )
    __proto__: Object
> module.exports
ƒ ( selector, context ) {
        // The jQuery object is actually just the init
constructor 'enhanced'
        return new jQuery.fn.init( selector, context,
rootjQuery );
    }
> module.exports();
init {}
    __proto__: Object(0)
>
```

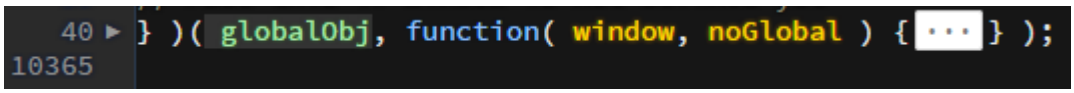
بعد فتح وحدة التحكم على الصفحة و جلب قيمة الكائن module.exports نجد أنه أصبح يشير إلى دالة بدل الكائن الذي كان يشير إليه سابقا، مما يعني أن الشرط الأول تحقق.

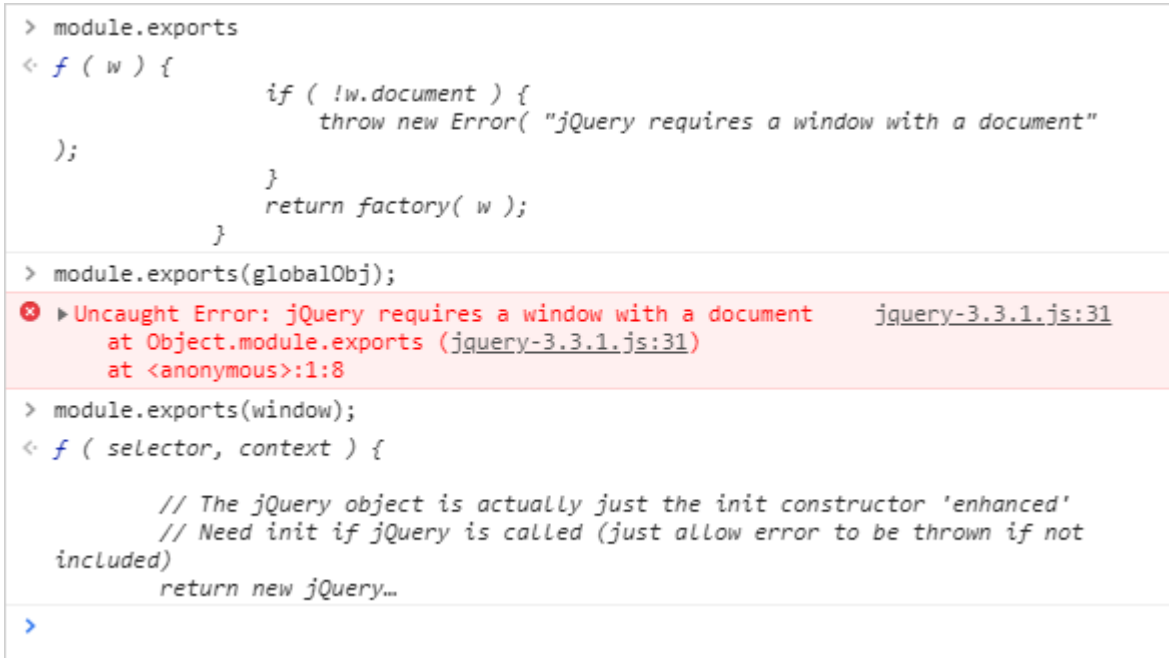
إن كان الشرط global.document خاطئاً (أي أن الكائن العام لا يحتوي على العقدة document)، فإن exports سيشير إلى الدالة المجهولة و التي تستقبل وسيطا واحدا، هذا الوسيط هو الكائن العام الذي سيمرره المبرمج ثم يستدعى الدالة exports لتتأكد بدورها من وجود الخاصية أو العقدة document ضمن الكائن الممرر لها، فإن لم يوجد فإنها سترمي خطأ، و إن وجد فإنها ترجع القيمة المرجعة من الدالة factory.

مثال للتوضيح:

```
<script type="text/javascript" >
    var module = {};
    module.exports = {};
    globalObj = {};
</script>
<script src="../frameworks/jquery-3.3.1.js" ></script>
```

أضفت هذه المرة الكائن globalObj الذي سأمرره ككائن عام للدالة المجهولة حتى أوضح الشرح السابق:





في حال لم يكن module و module.exports كائنان أو غير موجودان أصلا فهذا يعني أن المكتبة تتعامل مع بيئة المتصفح، و بالتالي سيتم تنفيذ else بالطبع و التي تقوم بإستدعاء الدالة factory مع تمرير المعامل الأول global (الكائن العام) إليها.

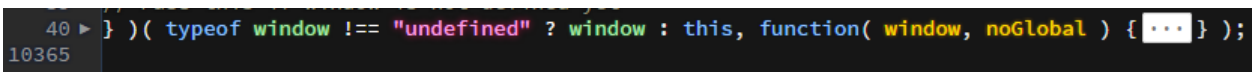
مثال للتوضيح:

أرجعت جميع الأمور إلى حالتها الطبيعية (الإفتراضية)!

ملف ال HTML:

```
<html dir="rtl">
    <head>
        <meta charset="utf-8">
    </head>
    <body>
        <script src="../frameworks/jquery-3.3.1.js"></script>
    </body>
</html>
```

مكتبة jQuery:



المعامل الأول (global): تم تمرير سطر برمجي يقوم بإرجاع الكائن العام.

```
typeof window !== "undefined" ? window : this,
```

الشرح بالتفصيل:

هذا السطر البرمجي عبارة عن معامل ثلاثي يقوم بالتحقق من صحة الوسيط الأول و الذي هو العبارة: typeof window !== "undefined"، التحقق هنا من أن نوع الكائن العام window لا يساوي القيمة "undefined"، فإن كان هذا صحيحا، فإن القيمة المرجعة هي الكائن العام window و إلا فإنه يتم إرجاع ما يشير إليه المعامل this، أي الكائن العام الخاص بالبيئة المعمول عليها. أيّا كانت القيمة المرجعة فهي ما سيتم تمريره كوسيط أول للدالة المجهولة.

لاحظ أنه تم إستعمال معامل عدم التساوي الصارم !== و ذلك لتجنب أي تحويل قسري قد يؤثر سلبا على النتائج.

لو طبقنا تلك العبارة في وحدة التحكم سترجع لنا الكائن العام:

```
> typeof window !== "undefined" ? window : this
  ▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent
  : Window, …}
>
```

إذن المكتبة تتعامل مع الكائن العام window. و بالتالي سنتعامل مع الدالة jQuery أو الإختصار $:

```
> jQuery
ƒ ( selector, context ) {

        // The jQuery object is actually just the init constructor 'enhanced'
        // Need init if jQuery is called (just allow error to be thrown if not
included)
        return new jQuery…
> $
ƒ ( selector, context ) {

        // The jQuery object is actually just the init constructor 'enhanced'
        // Need init if jQuery is called (just allow error to be thrown if not
included)
        return new jQuery…
> jQuery()
▶ jQuery.fn.init {}
> $()
▶ jQuery.fn.init {}
>
```

إذن كما قلنا سابقا أن الدالة المجهولة تقوم بالتحقق من البيئة التي يتم إستخدام مكتبة jQuery عليها. و ذلك لتأخذ قرارات محددة حول كيفية التعامل مع الكائن العام لهذه البيئة و إضافة خصائص المكتبة إليه حتى يتم الوصول إليها في أي سياق.

في المتصفح، مكتبة jQuery تتعامل مع الكائن العام window، و الذي يحتوي كل ما يخص النافذة المفتوحة و الصفحة، و بالخصوص شجرة الـ DOM حيث أن jQuery صممت للتعامل مع ال DOM.

المعامل الثاني (factory):

تم تمرير جسم دالة مجهولة، هذه الدالة هي التي تم تنفيذها خلال عمليات التحقق السابقة مع تمرير الكائن العام لها حسب البيئة المعمول عليها.

```
function( window, noGlobal ) { ··· }
```

هذه الدالة تحمل كل شيفرات مكتبة jQuery، و لقد قمت بطي هذه الدالة فقط للتسهيل الشرح.

إذن كل شيفرات مكتبة jQuery مغلفة داخل دالة مجهولة! هذا شيء جميل.

قبل أن ننغمس في جسم الدالة، دعنا نمسك في رأس الخيط:

في الأمثلة السابقة، عندما تم تنفيذ الدالة المجهولة التي تحمل هيكل المكتبة، كانت النتيجة المرجعة كالتالي:

```
> jQuery()
< ▶ jQuery.fn.init {}
```

يبدوا أن النتيجة التي ترجعها الدالة jQuery هي كائن من النوع jQuery.fn.init و ليس jQuery فقط!

إذن الدالة البانية هي الوظيفة init التابعة للكائن fn و الذي بدوره يعتبر خاصية تابعة للدالة jQuery!

إذا بحثنا في الدالة التي تغلف شيفرات المكتبة، سنجد أنها تعرّف مجموعة من المتغيرات أولا، ما يهمنا هو هذا السطر البرمجي:

```
131 var
132     version = "3.3.1",
133
134     // Define a local copy of jQuery
135     jQuery = function( selector, context ) {
136
137         // The jQuery object is actually just the init constructor 'enhanced'
138         // Need init if jQuery is called (just allow error to be thrown if not included)
139         return new jQuery.fn.init( selector, context );
140     },
```

تم تعريف متغير يحدد إصدار المكتبة، و بعده المتغير jQuery، هذا المتغير عبارة عن دالة تأخذ وسيطين selector و context، المفاجأة هي أن هذه الدالة لا تحمل الكثير من الشيفرات بل فقط تقوم بإرجاع كائن عن طريق إستدعاء دالة بانية jQuery.fn.init بإستعمال المعامل new!

الكائن الذي رأيناه سابقا تم بناؤه من الدالة البانية init التابعة للكائن fn و الذي يتبع بدوره إلى الدالة jQuery.

إذن jQuery ليست دالة بانية و إنما هي دالة عادية و لهذا تجدنا لا نستعمل المعامل new، هذه حيلة رائعة! حيث تجنبنا أن نكتب المعامل new كل مرة، فربما قد ننساه و بالتالي نحصل على أخطاء نحن في غنى عنها.

حسنا، لننزل قليلا و نرى ما الجديد:

```
145
146 ▼ jQuery.fn = jQuery.prototype = {
147
148       // The current version of jQuery being used
149       jquery: version,
150
151       constructor: jQuery,
152
153       // The default length of a jQuery object is 0
154       length: 0,
155
156 ▶     toArray: function() { ··· },
159
160       // Get the Nth element in the matched element set OR
161       // Get the whole matched element set as a clean array
162 ▶     get: function( num ) { ··· },
172
173       // Take an array of elements and push it onto the stack
174       // (returning the new matched element set)
175 ▶     pushStack: function( elems ) { ··· },
```

في السطر 146 نجد أنه تم تعريف الخاصية fn و نرى أنه أُسنِد إليها النموذج المبدئي للدالة jQuery، و مع ذلك تم تعيين كائن جديد كنموذج مبدئي يحمل خاصيتين (jquery, length) و 14 وظيفة فقط! (constructor, to Array, get ...)، لك أن تكتشف ما الذي تقوم به هاته الوظائف. و لكن من أين أتت المجموعة الهائلة من الوظائف التي توفرها المكتبة؟ حسنا، لو نزلنا قليلا و خصوصا بعدما ينتهي تعريف كائن النموذج المبدئي أي في السطر 227 سنجد أنه تم إضافة وظيفة جديدة (extend) إلى الدالة jQuery و إلى الكائن fn التابع لها:

```
226
227 ▼ jQuery.extend = jQuery.fn.extend = function() {
228       var options, name, src, copy, copyIsArray, clone,
229           target = arguments[ 0 ] || {},
230           i = 1,
231           length = arguments.length,
232           deep = false;
233
234       // Handle a deep copy situation
235 ▶     if ( typeof target === "boolean" ) { ··· }
242
243       // Handle case when target is a string or something (possible in deep copy)
244 ▶     if ( typeof target !== "object" && !isFunction( target ) ) { ··· }
247
248       // Extend jQuery itself if only one argument is passed
249 ▶     if ( i === length ) { ··· }
253
254 ▶     for ( ; i < length; i++ ) { ··· }
291
292       // Return the modified object
293       return target;
294   };
```

هذه الوظيفة تعتبر ممول للمكتبة، و من إسمها نجد أنها تعني التوسيع أي أن وظيفتها توسيع المكتبة، و حتى نرى آلية عملها، لنرى الخوارزمية القائمة عليها:

أولا تعرِّف الدالة مجموعة من المتغيرات من بينها المتغير target (أي الهدف) الذي سيشير إلى العنصر الأول في الشبه مصفوفة arguments و إلا فسيشير إلى كائن فارغ، (لاحظ إستعمال المعامل || لضبط الحالة الإفتراضية). إذن الدالة تتوقع تمرير وسائط!

أيضا المتغير length الذي يخزن عدد عناصر الشبه مصفوفة arguments.

بعد تعريف المتغيرات هناك ثلاث عمليات للتحقق و حلقة for ثم عملية إرجاع للهدف، أي المتغير target.

لقد طويت الشيفرات حتى يتسنى لنا أخذ فكرة عامة على الوظيفة extend. لنوسع كل واحدة و نرى ما بداخلها:

عملية التحقق الأولى:

```
233
234       // Handle a deep copy situation
235 ▼     if ( typeof target === "boolean" ) {
236           deep = target;
237
238           // Skip the boolean and the target
239           target = arguments[ i ] || {};
240           i++;
241       }
```

في عملية التحقق الأولى نجد أنه يتم التحقق من نوع المتغير target، أي الوسيط الممرر للوظيفة extend، إذا كان نوعه منطقي أي true أو false فإنه سيتم تعيين قيمة ذاك الوسيط إلى المتغير deep الذي تم تعريفه في الأعلى بالقيمة false.

بعد ذلك يتم إعادة تعيين المتغير target من جديد ليأخذ قيمة العنصر الموالي، فلو تلاحظ في سطر تعريف المتغيرات نجد المتغير i الذي يحمل القيمة 1، إذن هذا المتغير عبارة عن عداد، بعد إعادة تعيين قيمة المتغير target يتم زيادة قيمة العداد بواحد.

عملية التحقق الثانية:

```
242
243     // Handle case when target is a string or something (possible in deep copy)
244 ▼   if ( typeof target !== "object" && !isFunction( target ) ) {
245         target = {};
246     }
```

في هذه المرحلة يتم التحقق من أن المتغير target ليس كائنا و ليس دالة أيضا (لاحظ إستعمال الدالة isFunction و التي من يظهر إسمها أنها تتحقق من كون الوسيط الممر لها عبارة عن دالة أم لا!)

في حال كان ذلك صحيحا، فإنه يتم إعادة تعيين المتغير target ليشير إلى كائن فارغ.

لاحظ التعليق يقول: التعامل مع الحالة عندما يكون الهدف عبارة عن سلسلة نصية أو شيئ آخر. (التعليقات تساعدنا على فهم مسبق للعملية قبل أن نغوص فيها).

لما لا نقوم بالبحث عن الدالة isFunction و نرى طريقة عملها؟

سنجد أن الدالة قد تم تعريفها في السطر 74:

```
73
74 ▼ var isFunction = function isFunction( obj ) {
75
76       // Support: Chrome <=57, Firefox <=52
77       // In some browsers, typeof returns "function" for HTML <object> elements
78       // (i.e., `typeof document.createElement( "object" ) === "function"`).
79       // We don't want to classify *any* DOM node as a function.
80       return typeof obj === "function" && typeof obj.nodeType !== "number";
81   };
```

إذن الدالة تستقبل وسيط واحد ثم تقوم بإرجاع قيمة منطقية true أو false هذه القيمة ناتجة عن السطر التالي:

```
typeof obj === "function" && typeof obj.nodeType !== "number";
```

إذا كان نوع الوسيط عبارة عن دالة و "نوع العقدة" ليس رقما، فسيتم إرجاع القيمة true! هذا أمر متعمّد، فالعقد عبارة عن كائنات و نوعها دائما يكون رقما، لكن التحقق من هاته الجزئية سببه موضح في التعليقات أعلاه، حيث تم التنويه إلى المتصفحات المدعومة، و أيضا التنويه إلى أنه في بعض المتصفحات ترجع الدالة أو المعامل typeof القيمة "function" بالنسبة لعناصر HTML، و هذا خاطئ (لاحظ المثال المضمّن)، و نحن لا نريد أن تُصنّف العُقد على أساس أنها دوال. و لهذا وجب التحقق من هذا.

في حال كان الوسيط عبارة عن دالة، فإن الشطر الأول من التحقق سيكون صحيحا، أما بالنسبة للشطر الثاني، فإن الخاصية nodeType غير متوفرة لدى الدوال، و بهذا سيُرجع المعامل typeof القيمة "undefined"، و بالطبع "undefined" لا تساوي "number" و هذا صحيح، و بالتالي: true && true ترجع true.

عملية التحقق الثالثة:

```
247
248     // Extend jQuery itself if only one argument is passed
249     if ( i === length ) {
250         target = this;
251         i--;
252     }
```

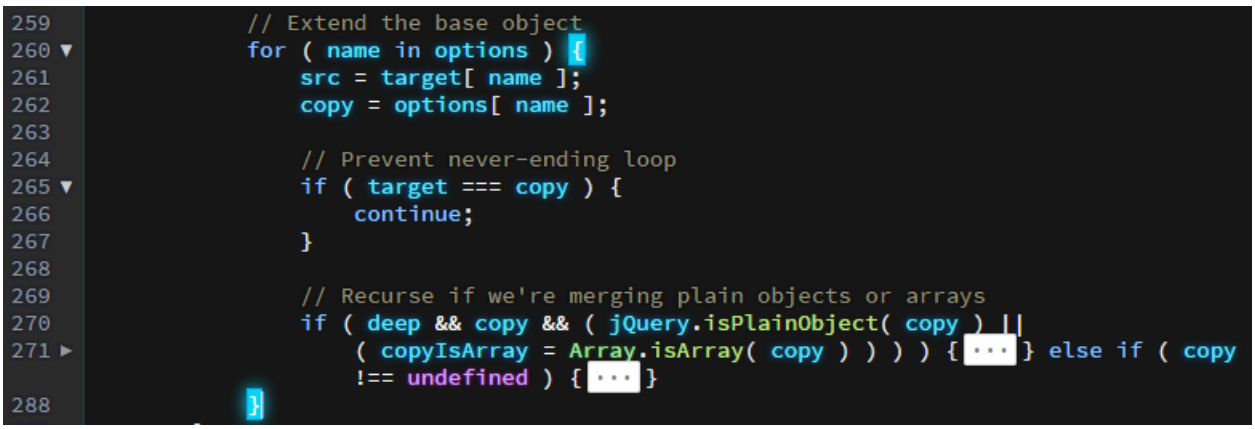
من خلال التعليق نفهم أن هذه العملية ستقوم بتوسيع المكتبة نفسها في حال تم تمرير معامل واحد، و ذلك من خلال مقارنة قيمة العداد i مع طول الشبه مصفوفة length.

الحلقة for:

```
253
254     for ( ; i < length; i++ ) {
255
256         // Only deal with non-null/undefined values
257         if ( ( options = arguments[ i ] ) != null ) {
258
259             // Extend the base object
260             for ( name in options ) { ··· }
289         }
290     }
```

نلاحظ أن هذه الحلقة تعتمد على العداد السابق i، و تقوم بعملية تحقق. هنا ضُرب طيرين بحجر واحد، إذ يسند العنصر المحدد في الشبه مصفوفة arguments إلى المتغير options أولا، و ثانيا التحقق من أن هذه القيمة لا تساوي null أو undefined.

بعد عملية التحقق يتم الدخول في حلقة ثانية:



```
259             // Extend the base object
260             for ( name in options ) {
261                 src = target[ name ];
262                 copy = options[ name ];
263
264                 // Prevent never-ending loop
265                 if ( target === copy ) {
266                     continue;
267                 }
268
269                 // Recurse if we're merging plain objects or arrays
270                 if ( deep && copy && ( jQuery.isPlainObject( copy ) ||
271                     ( copyIsArray = Array.isArray( copy ) ) ) ) { ··· } else if ( copy
                    !== undefined ) { ··· }
288             }
```

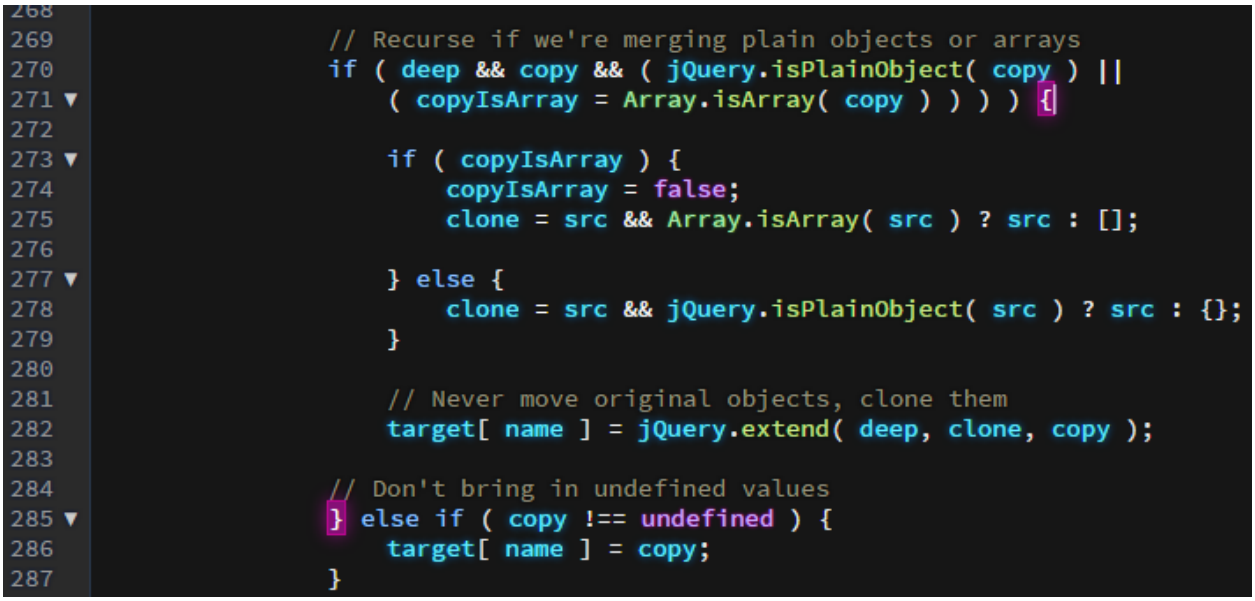
هذه الحلقة مسؤولة عن التوسعة، حيث تدور حول عناصر المتغير options.

تقوم بإسناد قيمة العنصر ذي الفهرس name و الموجود في target إلى المتغير src الذي يعني المصدر.

نفس العملية مع المتغير copy.

يتم التحقق ما إذا كان target هو نفسه copy أي يشيران إلى نفس الكائن. حيث سيتم تجاهل باقي الشيفرة و الإنتقال إلى العنصر الثاني في المتغير options، هذه الحركة تعمل على منع الوقوع في حلقة غير منتهية، ففي حال لم تعالج هذه المشكلة ستبقى الحلقة تضيف نفس العناصر من و إلى الكائن نفسه دون توقف.

لننتقل إلى عملية التحقق الأخرى:



```
269             // Recurse if we're merging plain objects or arrays
270             if ( deep && copy && ( jQuery.isPlainObject( copy ) ||
271                 ( copyIsArray = Array.isArray( copy ) ) ) ) {
272
273                 if ( copyIsArray ) {
274                     copyIsArray = false;
275                     clone = src && Array.isArray( src ) ? src : [];
276
277                 } else {
278                     clone = src && jQuery.isPlainObject( src ) ? src : {};
279                 }
280
281                 // Never move original objects, clone them
282                 target[ name ] = jQuery.extend( deep, clone, copy );
283
284             // Don't bring in undefined values
285             } else if ( copy !== undefined ) {
286                 target[ name ] = copy;
287             }
```

يتم التحقق هنا من المتغير deep و copy و نوعه إما كائن أو مصفوفة:

في حال كان تم تمرير true أو flase للطريقة extend كوسيط أول سيأخذ المتغير deep تلك القيمة و بناءً عليه يمكن أن تعمل العبارة if أعلاه.

يتم إسناد ناتج المعامل الثلاثي إلى المتغير clone حسب كل حالة.

بعد ذلك تعيين ما ترجعه الطريقة extend كقيمة للخاصية الجديدة المعرفة على الهدف target.

بالنسبة لإستدعاء الطريقة extend في السطر 282 سيتم نسخ ما في المتغير copy إلى المتغير clone و الذي هو الهدف في تلك الحالة حيث سيتم إرجاعه في الأخير، لأن الطريقة extend ترجع الهدف target.

في حال لم يتحقق الشرط الأول، ينتقل المحرك إلى العبارة else if ليتحقق من أن قيمة المتغير ليست undefined حيث لا نريد أن تضاف خصائص بقيم undefined إلى الكائن الهدف.

و في آخر المطاف يتم إرجاع الهدف target.

خلاصة الأمر في حال مررنا كائنا\مصفوفة واحدة إلى الطريقة extend تقوم بنسخ عناصر ذاك الوسيط إلى مكتبة الجي كويري، أي أنها تقوم بتوسيع المكتبة نفسها.

و في حال مررنا كائنين فأكثر يصبح الأول الكائن الهدف الذي ستضاف إليه الخصائص و الطرق الموجودة في الكائنات الأخرى.

هذه الحيلة رائعة تسمح لك بتوسيع المكتبة بالخصائص و الطرق التي تريدها، مما يضفي عليها بعض المرونة.

و من هذه النقطة ستجد أن المكتبة تعتمد على هذه الطريقة لتوسع نفسها، تحقق من ذلك بنفسك لترسخ لديك الفكرة و لو بحثت عن الطريقة isPlainObject ستجد أنها أضيفت عن طريق التوسيع!

بعد إنتهاء تعريف الوظيفة extend تم إستدعائها مباشرة و لاحظ أنه تم تمرير وسيط واحد فقط عبارة عن كائن! مما يعني أنه سيتم توسيع المكتبة ذاتها. هذا جيد، حاول أن تتأمل في تلك الوظائف المضافة و أفهم آلية عملها.

لننزل إلى الأسفل قليلا، سنجد إستدعاءً للوظيفة each و تعريف دالة isArrayLike، لن نتطرق إلى شرحها، بل ما يهمنا هو المتغير Sizzle الذي تم تعريفه بعدها، لاحظ التعليق المدرج تحته

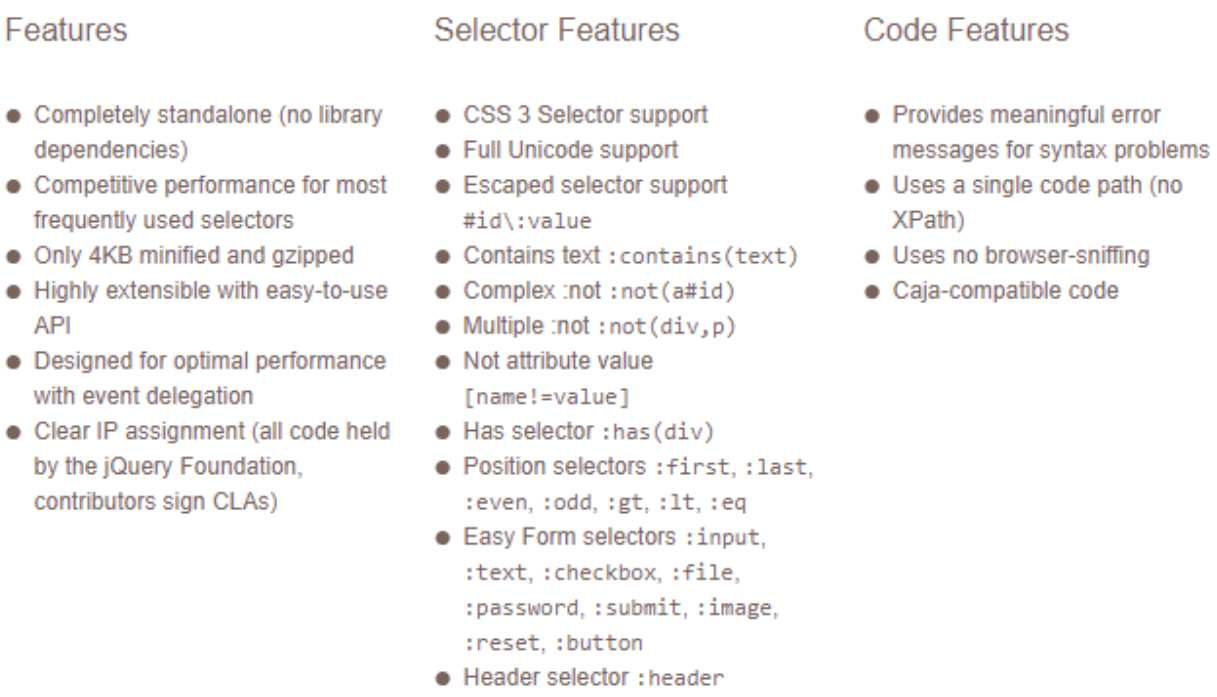
```
500
501  var Sizzle =
502  /*!
503   * Sizzle CSS Selector Engine v2.3.3
504   * https://sizzlejs.com/
505   *
506   * Copyright jQuery Foundation and other contributors
507   * Released under the MIT license
508   * http://jquery.org/license
509   *
510   * Date: 2016-08-08
511   */
512  (function( window ) {...})( window );
2755
```

يقول التعليق بأن Sizzle عبارة عن محرك محددات CSS و تم إدراج رابط لموقع sizzlejs، لنعرف أكثر عن هذا المحرك:

قبل أن ننتقل إلى هذا الموقع، نلاحظ أنه تم إسناد "دالة حالية التنفيذ" إلى المتغير Sizzle، و تم تمرير الكائن العام window إليها.

بعد زيارتنا لهذا الموقع سنجد أن Sizzle في الأصل مكتبة جافا سكريبت أخرى، فهل كنت تدري أن jQuery تحتوي على مكتبة أخرى كاملة بداخلها؟ حسنا، أنت الآن على دراية بذلك،

و كما ترى فإن المكتبة مُعدّة خصيصا ليتم إدراجها أو إستضافتها في مكتبات أخرى كمكتبتنا الحالية jQuery. يمكنك تنزيل ملف المكتبة لتطّلع عليه:



هذه المكتبة هي المسؤولة عن توفير ميزة الوصول إلى عناصر DOM بإستخدام أسلوب لغة CSS،

أي أنك تستعمل المحددات كما في CSS للوصول إلى عناصر الصفحة؛ هذا أمر رائع حقا، و هو مما ساعد مكتبة jQuery على إكتسابها تلك الشهرة.

إذن تعلمنا مما سبق أنه يمكن أن تصنع مكتبة مُعدّة للإستضافة في مكتبات أخرى، أو يمكن أن تضع مكتبة بأكملها داخل مكتبتك، و أيضا يمكن أن تضع تعبير دالة "حالية التنفيذ" داخل تعبير دالة أخرى "حالية التنفيذ".

هذه مهارات مهمة جدا تعلمناها من هذه الأسطر، قلّما تجد من يشرحها في أغلب دورات جافا سكريبت لولا تعمقك في المكتبات و الشيفرات التي يكتبها الآخرون.

حسنا، لنواصل تعلم المزيد!

دعنا لا نتعمق في مكتبة Sizzle، و لكن لو نظرنا في نهايتها سنجد أنها تقوم بإرجاع الدالة Sizzle المعرفة في مكان ما داخل المكتبة بعد القيام ببعض العمليات عليها (إبحث على تعريف الدالة و تأمله):

```
2751
2752  return Sizzle;
2753
2754  })( window );
2755
2756
2757
2758  jQuery.find = Sizzle;
2759  jQuery.expr = Sizzle.selectors;
2760
2761  // Deprecated
2762  jQuery.expr[ ":" ] = jQuery.expr.pseudos;
2763  jQuery.uniqueSort = jQuery.unique = Sizzle.uniqueSort;
2764  jQuery.text = Sizzle.getText;
2765  jQuery.isXMLDoc = Sizzle.isXML;
2766  jQuery.contains = Sizzle.contains;
2767  jQuery.escapeSelector = Sizzle.escape;
2768
```

بعد إكتمال تعريف المكتبة Sizzle أو تعبير الدالة "حالية التنفيذ"، تم القيام بضبط إشارات مرجعية في مكتبة jQuery إلى أهم المتغيرات التي تحتوي عليها مكتبة Sizzle. هذا جيد لنستمر!

لو نزلنا سنجد المزيد من الخصائص و إستعمال آخر للوظيفة extend من أجل إضافة المزيد من الخصائص، لنصل إلى هاته الأسطر:

تهيئة كائن jQuery!

```
2900
2901  // Initialize a jQuery object
2902
2903
2904  // A central reference to the root jQuery(document)
2905  var rootjQuery,
2906
2907      // A simple way to check for HTML strings
2908      // Prioritize #id over <tag> to avoid XSS via location.hash (#9521)
2909      // Strict HTML recognition (#11290: must start with <)
2910      // Shortcut simple #id case for speed
2911      rquickExpr = /^(?:\s*(<[\w\W]+>)[^>]*|#([\w-]+))$/,
2912
2913 ▶    init = jQuery.fn.init = function( selector, context, root ) { ··· };
3011
3012  // Give the init function the jQuery prototype for later instantiation
3013  init.prototype = jQuery.fn;
3014
3015  // Initialize central reference
3016  rootjQuery = jQuery( document );
3017
```

هنا نعثر على عملية التهيئة لكائن jQuery و أيضا الوظيفة init التي تم إستعمالها كدالة بانية في بداية المكتبة:

```
135     // Define a local copy of jQuery
136 ▼   jQuery = function( selector, context ) {
137
138         // The jQuery object is actually just the init constructor 'enhanced'
139         // Need init if jQuery is called (just allow error to be thrown if not included)
140         return new jQuery.fn.init( selector, context );
141     },
```

إذن هذه هي الوظيفة التي أربكتنا ربما، لنلقي نظرة عليها و نرى كيف تعمل:

```
2913 ▼     init = jQuery.fn.init = function( selector, context, root ) {
2914         var match, elem;
2915
2916         // HANDLE: $(""), $(null), $(undefined), $(false)
2917 ▼       if ( !selector ) {
2918             return this;
2919         }
2920
2921         // Method init() accepts an alternate rootjQuery
2922         // so migrate can support jQuery.sub (gh-2101)
2923         root = root || rootjQuery;
2924
2925         // Handle HTML strings
2926 ▶       if ( typeof selector === "string" ) { ··· } else if ( selector.nodeType ) {
2995             this[ 0 ] = selector;
2996             this.length = 1;
2997             return this;
2998
2999         // HANDLE: $(function)
3000         // Shortcut for document ready
3001         }
3002 ▶       else if ( isFunction( selector ) ) { ··· }
3009
3010         return jQuery.makeArray( selector, this );
3011     };
```

هذه هي الدالة البانية، حيث قمت بطي الشيفرات من أجل أن نأخذ نظرة عامة على الدالة:

تستقبل هاته الدالة ثلاث وسائط selector، context و root.

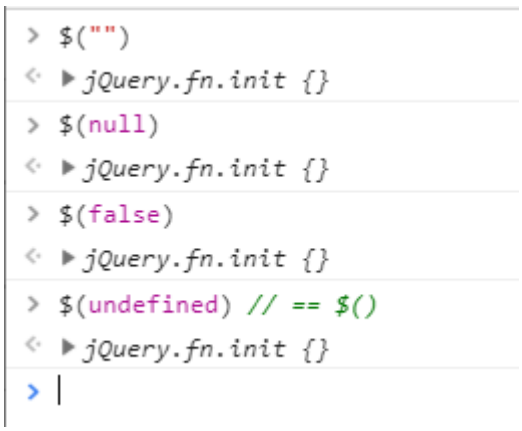
تأخذ المحدد و هو تلك السلسلة النصية التي نمررها بأسلوب لغة CSS إلى jQuery() أو $() (يشيران إلى نفس الدالة)، هذا المحدد يستعمل للوصول إلى عناصر DOM. مثلا:

$(".nav-bar")

تقوم بالتحقق من الحالات الممكنة التالية:

$(""), $(null), $(undefined), $(false)

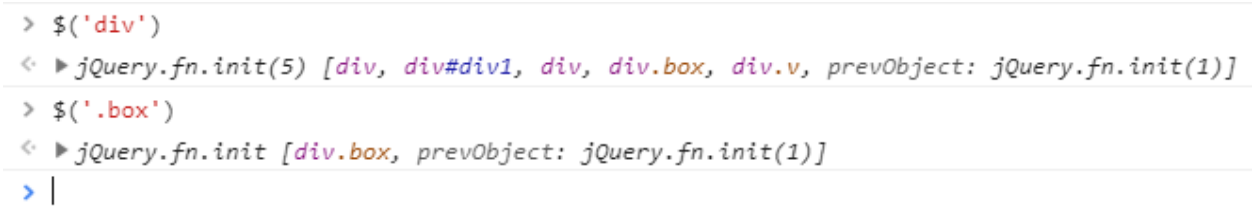
إذا كان المحدد إحدى هاته الحالات سيتم إرجاع الكائن الجديد المنشأ عبر الدالة البانية jQuery.fn.init(). مثال:



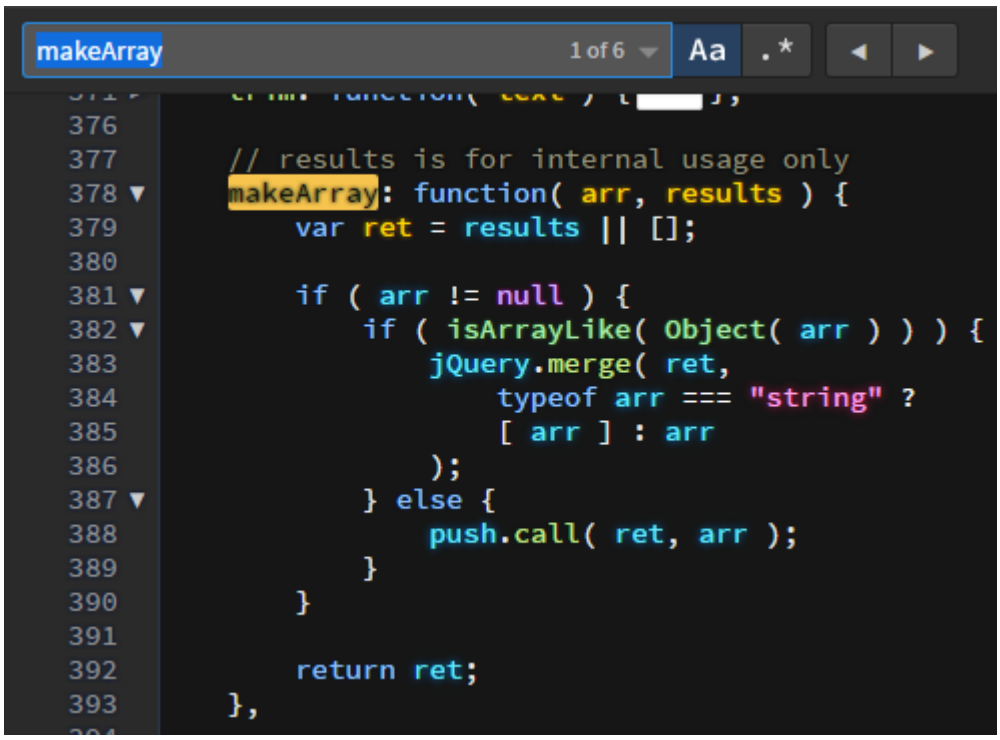
يتبقى معالجة الحالات الأخرى حين يتم تمرير سلسلة نصية أو عقدة أو دالة.

في الأخير تقوم الدالة بإرجاع مصفوفة عن طريق إستدعاء الوظيفة makeArray() مع تمرير المحدد و المعامل this الذي يشير إلى الكائن الجديد المنشأ عند إستدعاء الوظيفة jQuery.fn.init() بإستخدام المعامل new!

هذا ما نحصل عليه عند الوصول إلى العناصر عن طريق jQuery، مثال:



لما لا نبحث على الوظيفة makeArray و نعرف ما الذي ستفعله بذاك الكائن الجديد؟



ها هي ذي الوظيفة makeArray التي تأخذ الكائن الجديد المنشأ عبر الدالة البانية init، و الذي سيحل محل الوسيط results، تقوم بتحويله أو تضعه داخل مصفوفة ثم ترجعه لنا كنتيجة نهائية.

إذن تعلمنا مما سبق أنه لا بأس أن نرجع قيمة من الدالة البانية ما دامت ستكون ذاك الكائن الجديد نفسه بعدما نقوم ببعض العمليات عليه، لأن النموذج المبدئي سيضبط لهذا الكائن بنجاح.

حسنا، رأينا الدالة البانية jQuery.fn.init و ماذا تفعل، لنمضي قدما و نكتشف المزيد:

في السطر الذي يلي تعريف الوظيفة init هناك حيلة رائعة تم إستعمالها:

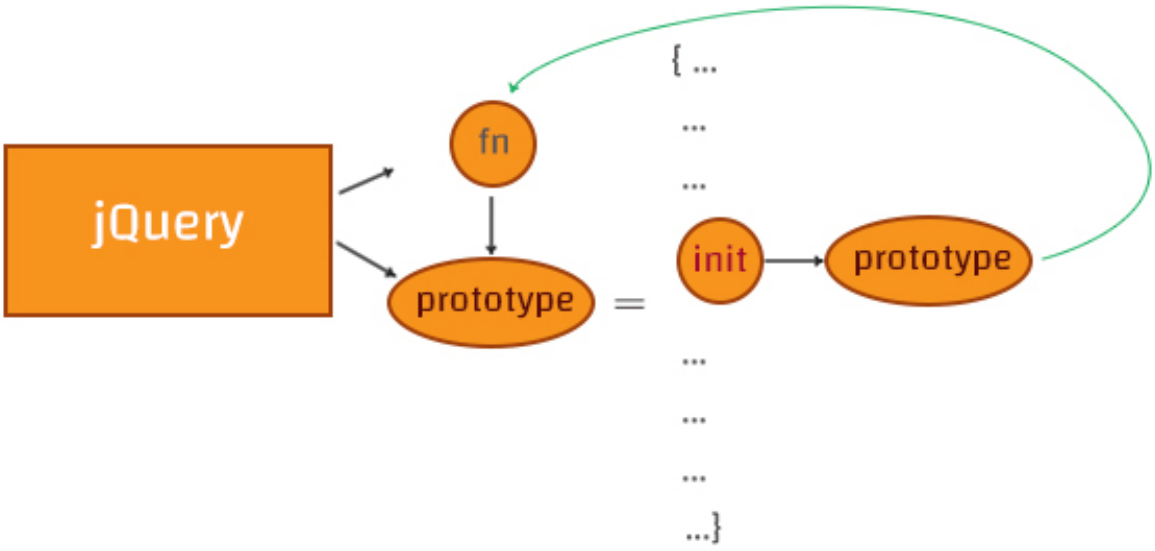
```
3012
3013  // Give the init function the jQuery prototype for later instantiation
3014  init.prototype = jQuery.fn;
3015
```

نلاحظ أنه تم ضبط النموذج المبدئي للدالة init ليشير إلى نفس النموذج المبدئي الخاص بالدالة jQuery، الدالة init تشير إلى نفس الوظيفة jQuery.fn.init

إذن أي كائن يتم إنشائه عبر الدالة البانية jQuery.fn.init سيرث جميع الخصائص و الوظائف الموجودة في النموذج المبدئي للدالة jQuery، أي يستطيع الوصول إليها.

إنها لحقا حيلة رائعة أن تصمم دالة تستدعي وظيفة بانية بداخل نموذجها المبدئي و ضبط النموذج المبدئي لهذه الوظيفة البانية ليشير إلى نفس النموذج المبدئي الخاص بالدالة الرئيسية أي نفس الكائن الذي تتواجد به الوظيفة البانية:

```
> jQuery.prototype
◂ ▶ Object [jquery: "3.3.1", constructor: ƒ, toArray: ƒ, get: ƒ, pushStack: ƒ, …]
> jQuery.prototype.init.prototype
◂ ▶ Object [jquery: "3.3.1", constructor: ƒ, toArray: ƒ, get: ƒ, pushStack: ƒ, …]
>
```



حسنا، لنواصل النزول و نكتشف المزيد.

تقريبا سنجد فقط إضافة لوظائف جديدة للمكتبة، و لكن هناك أمر مهم في بعض الوظائف.

دعني أوضح الأمر من جهة أخرى:

تتيح لنا مكتبة jQuery إستعمال أسلوب رائع، حيث يمكننا إستدعاء عدة وظائف متسلسلة بأسلوب التنقيط. مثال:

```
> $('.v').removeClass('v').addClass('nav-bar');
◂ ▶ jQuery.fn.init [div.nav-bar, prevObject: jQuery.fn.init(1)]
>
```

لاحظ إستدعاء الوظيفة addClass على الوظيفة removeClass، و هاته الأخيرة مستدعاة على الكائن الجديد المنشأ. لكن ألا يبدوا هذا خاطئاً؟ كيف نستدعى وظيفة على وظيفة أخرى و هي ليست تابعة لها؟ في الواقع السطر السابق صحيح و يؤدي وظيفته.

دعنا نرجع إلى الشيفرة المصدرية و نكتشف ما يجري:

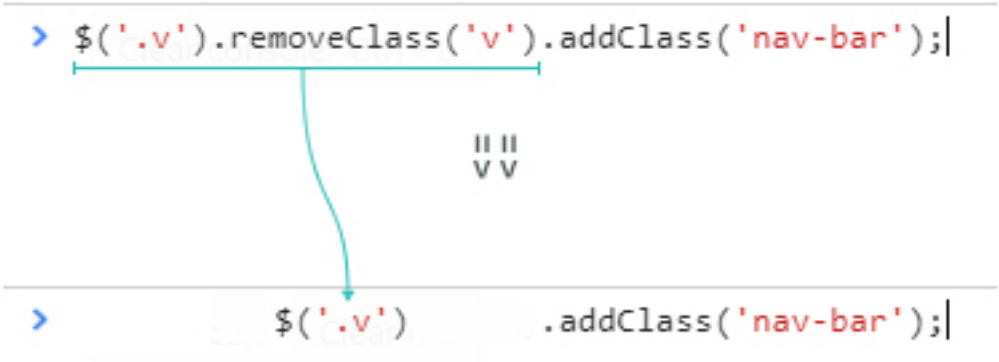
سيساعدنا البحث عن الوظيفة addClass مثلا إلى إكتشاف السر:

```
7807 ▼ jQuery.fn.extend( {
7808 ▼     addClass: function( value ) {
7809           var classes, elem, cur, curValue, clazz, j, finalValue,
7810               i = 0;
7811
7812 ▶         if ( isFunction( value ) ) {···}
7817
7818           classes = classesToArray( value );
7819
7820 ▶         if ( classes.length ) {···}
7841
7842           return this;
7843       },
```

دون التعمق في آلية عملها، يهمنا فقط ما تقوم بإرجاعه في الأخير، حيث نلاحظ أنها ترجع المتغير this. الأمر ذاته مع الوظيفة الثانية removeClass:

```
7845 ▼     removeClass: function( value ) {
7846           var classes, elem, cur, curValue, clazz, j, finalValue,
7847               i = 0;
7848
7849 ▶         if ( isFunction( value ) ) {···}
7854
7855 ▶         if ( !arguments.length ) {···}
7858
7859           classes = classesToArray( value );
7860
7861 ▶         if ( classes.length ) {···}
7886
7887           return this;
7888       },
```

هاتين الوظيفتين يتم إضافتهما إلى النموذج المبدئي fn للكائن الجديد المنشأ عن طريق الوظيفة init، مما يعني أن المتغير this سيشير إلي ذاك الكائن. بطريقة أوضح عند إستدعاء الوظيفة addClass على الكائن الجديد، ستقوم بعملها و ترجع ذاك الكائن لنا، و بهذا يمكن أن نستدعي الوظيفة removeClass عليه في نفس السطر:



يدعى مثل هذه بوظيفة التسلسل، حيث يتم إستدعاء وظيفة على أخرى وكل واحدة منهم تؤثر على الكائن الأب لأنها ترجع المتغير this الذي يشير إلى ذاك الكائن.

إذن تعلمنا من هذا أنه في حال أردنا أن نجعل الوظيفة تسلسلية كل ما علينا هو جعلها ترجع المتغير this.

حسنا، إلى هنا نكون قد تعلمنا حيلا رائعة و قوية جدا، و هذا يكفينا لنصنع مكتبتنا الخاصة بإستعمالها. أمر أخير: لنرى كيف يتم إطلاق مكتبة jQuery حتى نتمكن من إستعمالها:

```
10337  var
10338
10339      // Map over jQuery in case of overwrite
10340      _jQuery = window.jQuery,
10341
10342      // Map over the $ in case of overwrite
10343      _$ = window.$;
10344
10345  ▶ jQuery.noConflict = function( deep ) { ··· };
10356
10357  // Expose jQuery and $ identifiers, even in AMD
10358  // (#7102#comment:10, https://github.com/jquery/jquery/pull/557)
10359  // and CommonJS for browser emulators (#13566)
10360  ▼ if ( !noGlobal ) {
10361      window.jQuery = window.$ = jQuery;
10362  }
10363
10364
10365
10366
10367  return jQuery;
10368  } );
```

في السطور الأخيرة من المكتبة نجد عملية إطلاق أو إضافة المكتبة إلى الكائن العام window.

حيث يتم التحقق من عدم تمرير للوسيط noGlobal لإضافة المكتبة إلى الكائن العام window.
هل تتذكر الوسيط noGlobal في بداية المكتبة؟

قد تتسائل ما السبب من عملية الإضافة هاته؟ حسنا، المكتبة ككل مغلفة داخل دالة مجهولة و المتغيرات و الدوال المعرّفة داخل هذه الدالة مخزنة على مساحتها الخاصة، فكيف سنتمكن من الوصول إلى هذه المتغيرات و بالأخص الدالة jQuery؟ إذن لابد من إضافتها إلى الكائن العام حتى تصبح متوفّرة بشكل عام.

```
26          // See ticket #14549 for more info.
27          module.exports = global.document ?
28              factory( global, true ) :
29              function( w ) {
30                  if ( !w.document ) {
31                      throw new Error( "jQuery requires a window with a document" );
32                  }
33                  return factory( w );
34              };
35      } else {
36          factory( global );
37      }
38
39  // Pass this if window is not defined yet
40  } )( typeof window !== "undefined" ? window : this, function( window, noGlobal ) {
```

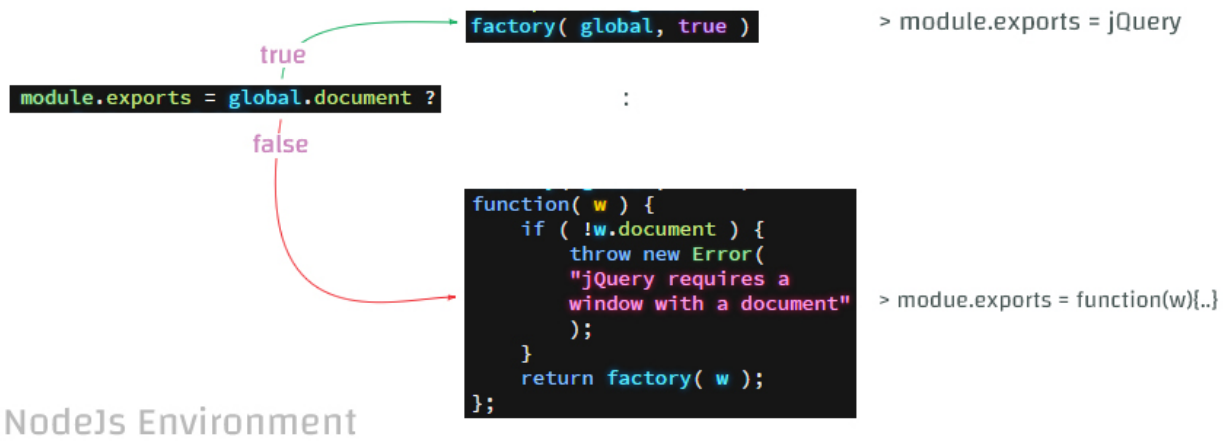
الدالة المجهولة الممررة كوسيط ثاني إلى الدالة الرئيسية "حالية التنفيذ" تستقبل وسيطين، الأول هو الكائن العام، و الثاني هو noGlobal، هذا الوسيط يُمرَّر فقط في حال كانت البيئة التي تشتغل عليها المكتبة nodeJs مثلا.

بالنسبة للوظيفة noConflict فإنها تسمح لك بوضع jQuery في متغير مختلف لأنها ترجعها في الأخير، و بتمرير القيمة true إليها فإن كل من jQuery و $ ستصبح قيمتهما undefined و ذلك بسبب أنهما أُضيفا فيما بعد إلى الكائن العام window و بالتالي فإن محاولة الوصول إلى خاصية غير موجودة يؤدي إلى إضافتها مع وضع القيمة undefined كقيمة مبدئية أو إفتراضية.

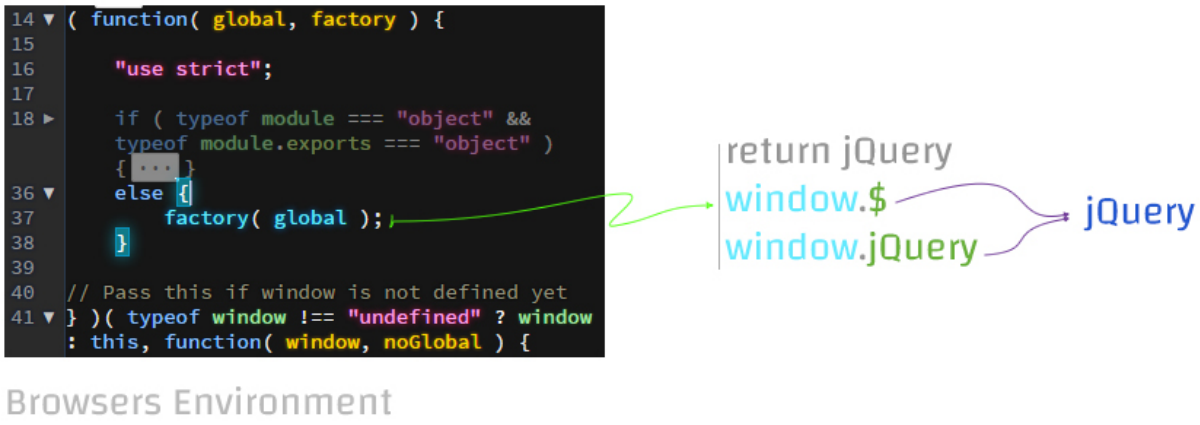
```
10341 jQuery.noConflict = function( deep ) {
10342     if ( window.$ === jQuery ) {
10343         window.$ = _$;
10344     }
10345
10346     if ( deep && window.jQuery === jQuery ) {
10347         window.jQuery = _jQuery;
10348     }
10349
10350     return jQuery;
10351 };
```

```
> $()
◂ jQuery.fn.init {}
> jQuery()
◂ jQuery.fn.init {}
> var ref = jQuery.noConflict(true);
◂ undefined
> ref()
◂ jQuery.fn.init {}
> $
◂ undefined
> jQuery
◂ undefined
>
```

توضيح بالنسبة لبيئة Node.js:



توضيح بالنسبة لبيئة المتصفح:



إذن هذه هي مكتبة jQuery ببساطة. ما هي إلا شيفرات جافا سكريبت! لذلك لا تخف من الإطلاع على الشيفرات المصدرية للمكتبات الشهيرة و أطر العمل، و أي شيفرات أخرى جيدة. يمكنك تعلم المزيد منها و هذا يساعدك في الإحتراف.

لقد أخذنا جولة سريعة داخل الشيفرة المصدرية لـ jQuery و اطّلعنا على بعض الحيل الرائعة و المفيدة حقا، خذ وقتك لتفهم أشياء أخرى و تعرّف على بعض الوظائف.

إنشاء مكتبة بسيطة

لنصنع مكتبتنا الخاصة!

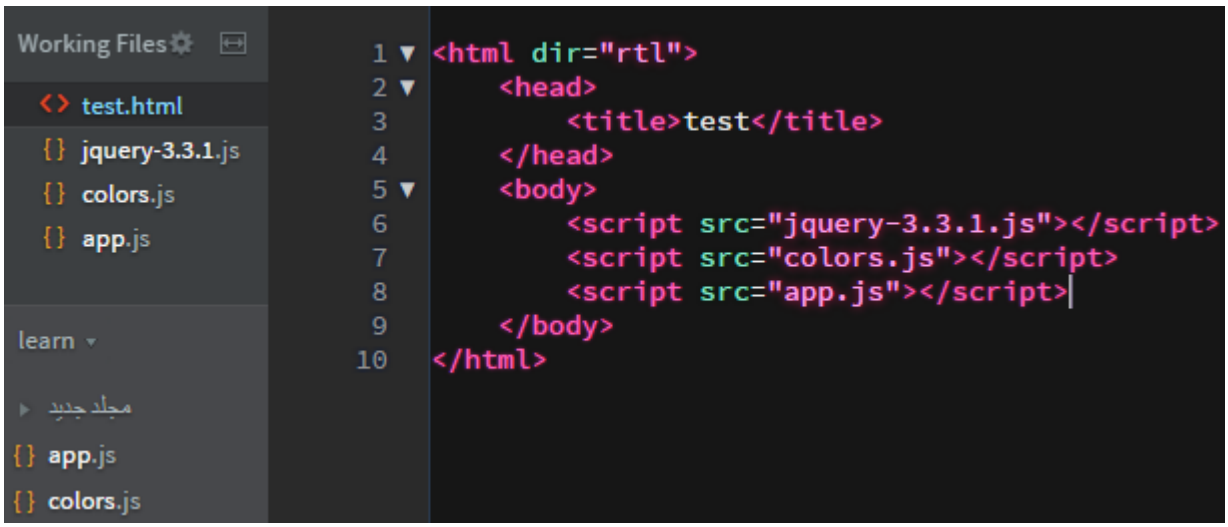
حسنا، حان الوقت لنستعمل كل ما تعلمناه سابقا، و نوظفه في مكتبة صغيرة خاصة بنا. و لنتفق على أننا لن نجعلها شاملة أو تعالج أمور معقدة، سنجعلها بسيطة جدا لأننا نريد التركيز على المفاهيم التي تعلمناها.

أولا دعنا نحدد متطلبات المكتبة:

- الهدف من المكتبة، سيكون هو ضبط اللون و الحدود لأي عنصر في شجرة ال DOM.
- الإسم: Colors
- سنعطيها أيضا إسما مختصرا هو الحرف c من أجل التمييز و التسهيل (مثل jQuery).
- تستقبل وسيطين أولهما هو اللون "إلزامي"، و الثاني هو عرض الحدود "إختياري".
- تدعم مكتبة jQuery. رغم أنها ستتعامل مع الألوان و الحدود إلا أن ما نريده هو أن تكون قابلة لإن تأخذ كائن jQuery الذي سيشير إلى العنصر المرغوب بتطبيق التغييرات عليه.
- تكون قابلة للإستعمال في مكتبات أخرى كما هو الحال مع Sizzle.

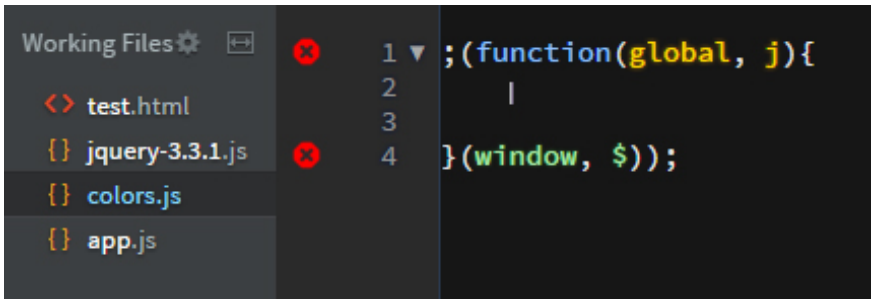
حسنا لقد حددنا المتطلبات، و قبل أن نبدأ الكتابة، سيكون لدينا ثلاث ملفات في هذا المشروع:

- ملف test.html.
- مكتبة jQuery (سأستخدم الإصدار 3.3.1، و يمكنك إستخدام أي إصدار تريد)
- ملف مكتبتنا الخاصة تحت مسمى colors.js.
- ملف app.js من أجل فحص مكتبتنا و تنفيذ التغييرات على الصفحة.



لنباشر الآن بكتابة شيفرة مكتبتنا:

سنرغب في أن نجعل شيفرة المكتبة آمنة إذ يمكن إستعمالها في أي مكتبة أخرى، لذلك نحتاج إلى سياق تنفيذ جديد نُعرّف به متغيراتنا لتبقى بشكل آمن، و نمرر لها ما نريد لكي تستطيع الوصول إليه. نريدها أن تصل إلى الكائن العام window و مكتبة jQuery. إذن سنغلفها في دالة حالية التنفيذ:



لاحظ علامة الفاصلة المنقوطة قبل قوسي التجميع، إنها حيلة أخرى و ذلك فقط لنضمن أن أي شيفرة تسبق شيفرة مكتبتنا ستنتهي بفاصلة منقوطة، فهي مجرد خطوة إحتياطية.

مررنا للدالة "حالية التنفيذ" الكائن العام window و مكتبة jQuery، و يمكن أن نحسّن الشيفرة لنجعلها تتحقق من الكائن العام الممرر إلى الدالة كما تفعل jQuery. ربما مستقبلا حين تكبر مكتبتنا، أما حاليا فلنجعل الأمور بسيطة.

لنضبط الآن الكائن Colors بالطريقة التي تستعملها مكتبة jQuery، إذ سنحاكي أسلوبها حيث سننشأ دالة تولد لنا كائن، لا نريدها أن تكون هي الدالة البانية و إنما ترجع كائنا من دالة بانية

أخرى، هذا حتى لا نضطر إلى إستعمال المعامل new في كل مرة نريد أن ننشئ فيها كائناً. أيضا نمرر لهذه الدالة قيمة اللون و عرض الحدود، أي أنها ستستقبل وسيطين:

```
;(function(global, j){

    // إنشاء كائن جديد.
    var Colors = function(color, border){
        return new Colors.init(color, border);
    }

}(window, $));
```

عرّفنا الدالة Colors و التي تستقبل وسيطين، و تقوم بإرجاع الكائن الجديد المنشأ عبر الوظيفة init التابعة للدالة Colors، حيث تستقبل الوظيفة init نفس الوسيطين الممرّرين إلى الدالة Colors.

لم أرد أن أضيف الخاصية fn كما هو الحال في jQuery، أردت أن تكون الشيفرة بسيطة جدا.

إذن لنعرّف الوظيفة init:

```
// تهيئة الكائن جديد.
Colors.init = function(color, border){
    // التأكد من تمرير اللون.
    // و إلا سيتم إطلاق خطأ.
    if(!color) throw 'color is undefined!';
    this.color = color;
    this.border = border ? border + 'px solid #000' : 'none';
};
```

الوظيفة init لتهيئة الكائنات الجديدة. حيث تقوم بالتحقق من أن الوسيط color تم تمريره و إلا فإنها سترمي خطأ.

تُسنِد اللون المُمرّر إلى الخاصية color و عرض الحدود إلى الخاصية border مع وضع الحالة الإفتراضية في حال عدم تمرير هذا الأخير. لاحظ إستعمالنا للمعامل الثلاثي بدل المعامل "أو" || و ذلك لأننا نريد أن يتم تخزين سلسلة نصية في كلتا الحالتين.

لنتحقق من أن مكتبتنا تعمل بشكل جيّد. لكن قبل هذا علينا أن نجعل المكتبة متوفرة بشكل عام، إذن يجب أن نضيفها إلى الكائن العام:

```
// إطلاق Colors & C
global.C = global.Colors = Colors;
```

بهذا نكون قد وفرنا إمكانية الوصول إلى المكتبة عن طريق إحدى المتغيرين إما Colors أو C بإختصار.

```
;(function(global, j){

    // إنشاء كائن جديد.
    var Colors = function(color, border){
        return new Colors.init(color, border);
    }

    // تهيئة الكائن جديد.
    Colors.init = function(color, border){
        // التأكد من تمرير اللون.
        // و إلا سيتم إطلاق خطأ.
        if(!color) throw 'color is undefined!';
        this.color = color;
        this.border = border ? border + 'px solid #000' : 'none';
    };

    // Colors & C إطلاق
    global.C = global.Colors = Colors;

}(window, $));
```

إذن لنتحقق الآن:

```
var myColor = C('#0f0');
var myColor2 = C('#00f', 5);
console.log(myColor);
console.log(myColor2);
```

في ملف app.js إستخدمنا مكتبتنا لإنشاء كائنين جديدين بقيم مختلفة، و طبعناهما إلى وحدة التحكم، و النتيجة كالتالي:

```
▶ Colors.init {color: "#0f0", border: "none"}
▶ Colors.init {color: "#00f", border: "5px solid #000"}
>
```

هذا رائع حقا! المكتبة تعمل بشكل جيد. الكائن الأول يحمل اللون الذي حددناه و تم وضع القيمة الإفتراضية للحدود. و الثاني يحمل اللون و الحدود.

حسنا ماذا عن الوراثة بالنسبة للكائن الجديد المنشأ؟ أي النموذج المبدئي لذاك الكائن؟ من الجميل أن تتوفر بعض الوظائف لهذا الكائن حتى نتمكن من إجراء عمليات عليه، و أيضا يكون للمكتبة فائدة.

سنجعل النموذج المبدئي هو نفسه الخاصية prototype التابعة للدالة Colors كما في jQuery. سيحمل هذا الكائن الوظائف التي نريدها أن تكون متوفرة لأي كائن جديد يتم إنشائه:

```
    // إنشاء كائن جديد.
    var Colors = function(color, border){
        return new Colors.init(color, border);
    }

    // النموذج المبدئي
    Colors.prototype = {};

    // تهيئة الكائن جديد.
    Colors.init = function(color, border){
        // التأكد من تمرير اللون.
        // و إلا سيتم إطلاق خطأ.
        if(!color) throw 'color is undefined!';
        this.color = color;
        this.border = border ? border + 'px solid #000' : 'none';
    };

    // ضبط النموذج المبدئي للكائن الجديد
    Colors.init.prototype = Colors.prototype;

    // Colors & C إطلاق
    global.C = global.Colors = Colors;
```

لنملئ هذا النموذج ببعض الوظائف. بالمناسبة، سنجعل كل وظيفة قابلة للتسلسل، فهل تتذكر كيف يتم ذلك؟ فقط علينا أن نرجع من كل وظيفة المتغير this الذي يشير إلى ذاك الكائن الجديد:

```
// وظيفة لتغيير اللون وهي قابلة للتسلسل .
changeColor: function(color){
    this.color = color;
    return this;
},
```

أضفنا الوظيفة changeColor التي تقوم بتغيير اللون، و هي قابلة للتسلسل.

دعنا نتحقق من هذا:

```
var myColor = C('#0f0');
console.log( myColor );
myColor.changeColo('#00d');
console.log( myColor );
```

```
▶ Colors.init {color: "#0f0", border: "none"}
▶ Colors.init {color: "#00d", border: "none"}
>
```

هذا جيد، إنها تعمل بشكل سليم. حسنا لنضف المزيد:

```
// "قابلة للتسلسل" .rgb تغيير اللون عبر نظام ال
rgbColor: function(r, g, b){
    this.color = "rgb(" + r + "," + g + "," + b + ")";
    return this;
},
```

الوظيفة rgbColor تقوم بتغيير اللون بنظام ال rgb، و هي قابلة لتسلسل أيضا.

لنتحقق من أن الوظيفتين قابلتين للتسلسل:

```
> myColor.changeColor('#ff0').rgbColor(43,12,100);
◂ ▸Colors.init {color: "rgb(43,12,100)", border: "none"}
> myColor.rgbColor(43,12,100).changeColor('#ff0');
◂ ▸Colors.init {color: "#ff0", border: "none"}
> myColor.changeColor('#000').changeColor('#fdce42');
◂ ▸Colors.init {color: "#fdce42", border: "none"}
> |
```

هذا جيد، الوظيفتان قابلتان للتسلسل. لنواصل إضافة الوظائف:

```
// وضع الحدود. "قابلة للتسلسل"
setBorder: function(w, s, c){
    if(typeof w !== "number") throw 'border width must be a number';
    s = styles.indexOf(s) !== -1 ? s : 'solid';
    c = c || '#000';
    this.border = w +'px ' + s + ' ' + c;
    return this;
},
```

الوظيفة setBorder لوضع حدود حول العنصر المرغوب مع إمكانية تخصيص عرض و نمط و لون الحدود، و هي قابلة للتسلسل أيضا. لاحظ إستعمالنا للمعامل "أو" || لتعيين الحالة الإفتراضية بالنسبة للون الحدود و إستعملت المعامل الثلاثي لتعيين القيمة 'solid' كقيمة إفتراضية في حال لم تكن القيمة الممرّرة موجودة في المصفوفة styles.

لقد عرّفت المصفوفة styles قبل النموذج المبدئي، حيث تحمل أنماط الحدود، و هذا من أجل التحقق أن النمط الممرّر متوفر في لغة css أم لا. هذه المصفوفة متوفرة للمكتبة فقط أي أنها معرفة في الدالة "حالية التنفيذ"، مما يجعلها غير متوفرة في السياق العام أو أي سياق تنفيذ خارجي:

```
var styles = ['dashed', "double", "dotted","groove", "inset", "outside",
"ridge", "solid"];
```

مثال:

```
> myColor.setBorder(2,'dashed','#eef');
◂ ▶ Colors.init {color: "#0f0", border: "2px dashed #eef"}
> myColor.setBorder(2,'dashed','#eef').changeColor('#fd013c');
◂ ▶ Colors.init {color: "#fd013c", border: "2px dashed #eef"}
>
```

```
// إزالة الحدود. "قابلة للتسلسل"
clearBorder: function(){
    this.border = 'none';
    return this
},
```

الوظيفة clearBorder تقوم بإزالة الحدود و هي قابلة للتسلسل.

لقد حددنا في المتطلبات أن مكتبتنا ستدعم مكتبة jQuery أي أننا سنوفر إمكانية الوصول إلى العناصر عن طريق إستخدام أسلوب jQuery في إحدى الوظائف. الوظيفة applyOn تسمح بتطبيق التغييرات على عنصر معين بإستخدام المحددات، هذه الوظيفة غير قابلة للتسلسل.

```
// تطبيق الإعدادات على العنصر المرغوب.
applyOn: function(selector){
    $(selector).css({"color": this.color, "border" : this.border });
}
```

سنكتفي بهذا العدد من الوظائف، و هذا هو الشكل النهائي لنموذجنا المبدئي:

```
// النموذج المبدئي
Colors.prototype = {
    // وظيفة لتغيير اللون وهي قابلة للتسلسل .
    changeColor: function(color){
        this.color = color;
        return this;
    },
    // "قابلة للتسلسل" .rgb تغيير اللون عبر نظام ال.
    rgbColor: function(r, g, b){
        this.color = "rgb(" + r + "," + g + "," + b + ")";
        return this;
    },
    // "قابلة للتسلسل" .وضع الحدود
    setBorder: function(w, s, c){
        if(typeof w !== "number") throw 'border width must be a number';
        s = styles.indexOf(s) !== -1 ? s : 'solid';
        c = c || '#000';
        this.border = w +'px ' + s + ' ' + c;
        return this;
    },
    // "قابلة للتسلسل" .إزالة الحدود
    clearBorder: function(){
        this.border = 'none';
        return this
    },
    // .تطبيق الإعدادات على العنصر المرغوب
    applyOn: function(selector){
        $(selector).css({"color": this.color, "border" : this.border });
    }
};
```

حسنا، إلى هنا نكون قد إنتهينا من ضبط مكتبتنا البسيطة جدا. لنتحقق من أنها سليمة و تؤدي وظائفها بشكل جيد:

أولا أعددت بعض العناصر في ملف html كالتالي:

```
<html dir="rtl">
    <head>
        <title>test</title>
        <style>
            p{ font-size: 20px;}
        </style>
    </head>
    <body>
        <h1>مرحبا بكم مع مكتبتي الصغيرة <span id="text">Colors</span>!</h1>
        <p><span>Colors</span> هي مكتبة بسيطة تضبط اللون و الحدود لأي عنصر نصي.</p>

        <script src="jquery-3.3.1.js"></script>
        <script src="colors.js"></script>
        <script src="app.js"></script>
    </body>
</html>
```

حسنا، لنحدد لونا و حدودا إلى العنصر ذي المعرف text:

```
var myColor = C('#fd013c',2);
myColor.applyOn('#text');
console.log(myColor);
```

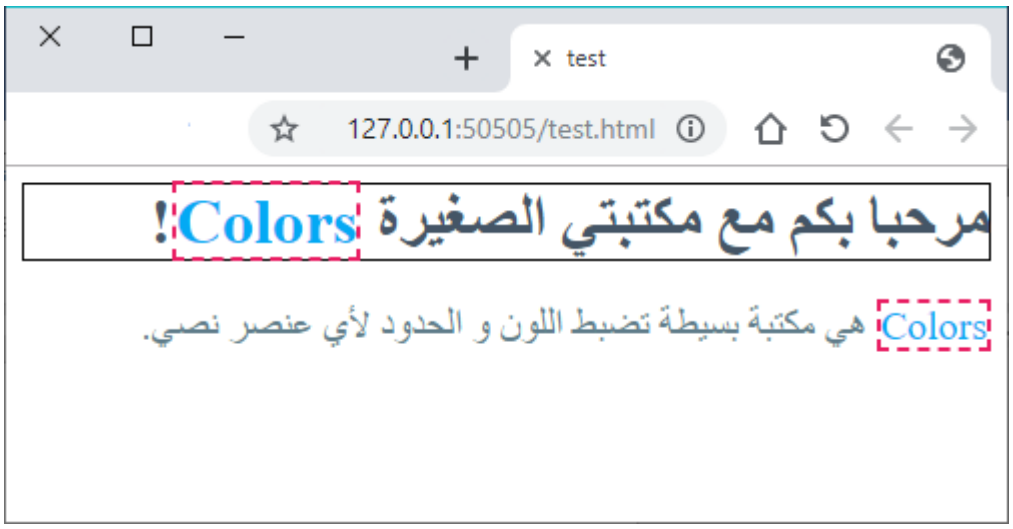
بعد تحديث الصفحة سنرى التغيير يظهر في الصفحة:


```
▶ Colors.init {color: "#fd013c", border: "2px solid #000"}
>
```

هذا رائع! لنطبق شيئا آخر:

```
var myColor = C('#456',1);
myColor.applyOn('h1');
myColor.changeColor('#607d8b').
        clearBorder().
        applyOn('p');
myColor.setBorder(2,'dashed', '#e91e63').
        changeColor('#03A9F4').
        applyOn('span');
console.log(myColor);
```

بعد تحديث الصفحة:



شاهد الروعة يا صديقي! أليست جافا سكريبت رائعة حقا!!!

وصلنا إلى نهاية مسيرتنا مع الجافا سكريبت، و قد عرجنا على أهم المفاهيم التي يجب أن تكون ملما بها حتى تتذوق حلاوة الجافا سكريبت و تتمتع بروعتها. لذا إنطلق إلى عالم الإحترافية و حاول صقل مهاراتك بمزيد من الإطلاع على الشيفرات المكتوبة بشكل جيد، و إستكشف مصادر المكتبات و أطر العمل و مارس ما تعلمته.

مرة أخرى: لا تخف! فالأمور أبسط مما تتخيل. تذكر أن هناك بعض الأساليب و الحيل التي يمكنك فعلها بلغة الجافا سكريبت بينما لا يمكنك ذلك بلغات أخرى.

تأكد أن الإبداع ليس له حدود، و بالطبع تستطيع أن تُخرج لهذا العالم إبداعاتك التي ربما ستكتسب شهرة كما إكتسبتها jQuery و غيرها من المكتبات، و لم لا؟ فليس هنالك مستحيل يا صديقي! الإصرار و العزيمة هما سر النجاح.

أرجو أن يكون هذا الكتاب إضافة نوعية جديدة و جيدة للمكتبة العربية و لك في مسيرتك البرمجية نحو الإحتراف أيضا.

إلى لقاء آخر مع إصدارات مستقبلية لهذا الكتاب أو كتابات أخرى. مع خالص تحياتي.

المؤلف.