تعلم
استخدام JavaScript للتعامل مع HTML DOM

عبد اللطيف ايمش

تعلم DOM

استخدام JavaScript
للتعامل مع مستندات HTML

ترجمة
عبد اللطيف ايمش

تقديم

يغفل الغالبية من المبرمجين عن أهمية تعلم التعامل مع مستندات HTML مباشرةً عبر JavaScript، دون استعمالهم لمكتبةٍ جاهزةٍ مثل jQuery؛ مما يورث قصورًا في فهمهم لطريقة عمل تلك المكتبات، ويرتبطون بها تمامًا مما يجعلهم غير قادرين على تعديل عناصر HTML دونها.

لذا أتى هذا الكتاب ليشرح كيفية الاستفادة من DOM لتعديل عناصر HTML عبر JavaScript شرحًا عمليًا مدعَّمًا بالأمثلة القابلة للتطبيق، والتي توضِّح المفاهيم البرمجية التي يحاول هذا الكتاب إيصالها، ويأتي في آخره فصلٌ نُنشِئ فيه مكتبةً شبيهةً بمكتبة jQuery تدريبًا عمليًا على استعمال دوال وكائنات DOM لتعديل المستندات.

هذا الكتاب مترجمٌ عن كتاب «DOM Enlightenment» لمؤلفه Cody Lindley، والذي نَشَرتَه دار نشر O'Reilly لاحقًا بنفس الاسم. نُشِرتَ هذه النسخة المترجمة بعد أخذ إذن المؤلف.

وفي النهاية، أحمد الله على توفيقه لي بإتمام العمل على الكتاب، وأرجو أن يكون إضافةً مفيدةً للمكتبة العربية، والله ولي التوفيق.

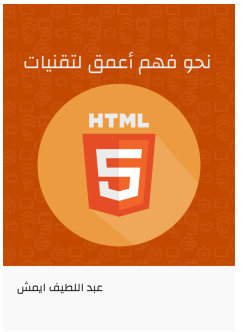
عبد اللطيف محمد أديب ايمش

حلب، سورية 2018/5/21

هذا الكتاب برعاية

وادي التقنية موقعٌ تقنيٌ عربيٌ يُعنى بتتبع أخبار البرمجيات الحرة والمواد التعليمية المتعلقة بها، يكتب فيه عدد من المتطوعين المهتمين بالبرمجيات الحرة والتقنية بشكل عام؛ يهتم وادي التقنية بمواضيع مثل أنظمة التشغيل الحاسوبية والهاتفية، ولغات البرمجة، والمكتبات البرمجية، وتقنيات الويب، وأخبار شركات البرمجة الكبرى، والمصادر المفتوحة، والعتاد وأجهزة الحاسوب .

كتب أخرى لعبد اللطيف ايمش



جدول المحتويات

تمهيد

هذا الكتاب ليس مرجعًا تفصيليًّا لميزات DOM أو JavaScript، لكنه ربما يكون أكثر الكتب شرحًا لطرائق استخدام ميزات DOM دون الحاجة إلى استعمال مكتبة أو إطار عمل. هنالك سببٌ وجيهٌ لقّلة المؤلفات عن هذا الموضوع، فأغلبية مؤلفي الكتب التقنية لا يرغبون بالغوص في هذا المجال بسبب الكم الهائل من الاختلافات في تطبيق المتصفحات القديمة لمواصفات DOM (DOM specifications) هذا في حال طبّقتها تلك المتصفحات من الأساس!

ولخدمة الغرض والهدف من هذا الكتاب (والذي هو استيعاب المفاهيم الأساسية استيعابًا كاملًا)، فسأنحّي الفوضى الموجودة في الواجهة البرمجية (API) التابعة للمتصفحات القديمة جانبًا بهدف إظهار الميزات الحديثة في DOM؛ أي أنني سأحاول تجنّب الخوض في الماضي (وتعقيدات الاختلافات بين المتصفحات) لأركّز على الحاضر والمستقبل. وعلى أيّة حال، هنالك حلولٌ جاهزٌ في متناولنا كمكتبة jQuery للتعامل مع المتصفحات القديمة، وأنصحك أن تستعمل مكتبةً مثل jQuery عندما تتعامل مع تلك المتصفحات.

وعلى الرغم من أنني لا أشجّع فكرة التخلي عن المكتبات وكتابة شيفرات DOM يدويًا، إلا أنني كتبتُ هذا الكتاب وفي نيتي أن أجعل المطورين يدركون أنَّ مكتبات التعامل مع DOM ليس ضرورية دومًا عند التعامل مع DOM. وألّفتُ هذا الكتاب أيضًا لِمَن يكتب شيفرات JavaScript لبيئةٍ وحيدةٍ (أقصد هنا متصفحًا وحيدًا، أو متصفحات الهواتف الذكية، أو أداةً لتحويل شيفرات HTML و CSS و JavaScript إلى تطبيقاتٍ للهواتف مثل كوردوفا [Cordova]). فربما ستجعل المعلومات -التي ستتعلمها في هذا الكتاب- من استخدام مكتبات DOM أمرًا غير ضروريٍ في الحالات المثالية.

1. من عليه قراءة هذا الكتاب

كتبتُ هذا الكتاب وأنا أفكّر بنوعين من المطورين، سأفترض أنَّ كلا النوعين ذوا إمكانياتٍ برمجية متوسطة أو متقدمة في لغات JavaScript و HTML و CSS.

أوّل نوعٍ هو مَن له درايةٌ كافيةٌ في JavaScript أو jQuery، لكنه لم ينفق وقتًا لفهم الغاية والهدف من المكتبات مثل jQuery، فذلك النوع من المطورين -بعد تسلُّحِه بالمعلومات التي أخذها من هذا الكتاب- يجب أن يقدر على فهم القيمة التي توفرها jQuery وكيف تساعد المطورين عند التعامل مع DOM، إضافةً إلى فهم طريقة «تجريد» jQuery لآلية التعامل مع DOM وتوحيد الواجهة البرمجية التي نتعامل فيها مع أكثر من متصفح، ولماذا (وكيف) تكمل jQuery الفراغات وتسد النواقص.

النوع الثاني هو المهندس الذي طُلِبَ منه كتابة مستندات HTML التي ستعمل في المتصفحات الحديثة فقط، أو تلك التي ستحوّل إلى شيفرات خاصة بأنظمة التشغيل (مثل كوردوفا) ويرغب في تجنّب تحمّل عبء استخدام مكتبةٍ ما.

2. التفاصيل التقنية والتجاوزات والمحدوديات

المحتوى والشيفرات الموجودة في هذا الكتاب مكتوبة في متصفحٍ حديث (أي IE9+، وآخر إصدار من Firefox و Chrome و Safari و Opera)، فغرضي هو شرح المفاهيم التي تعمل في المتصفحات الحديثة دون مكتبات أو إضافات (وتضمين أمثلة عليها). إذا جنحتُ عن هذا الهدف لسببٍ من الأسباب فسأبيّن ذلك للقارئ كي يتنبّه. أحاول عادةً أن أبتعد ما استطعت عن تضمين

أيّة معلومـــات في هـــذا الكتـــاب تخصّ متصـــفحًا معينًـــا أو تُطبَّق في نســـبةٍ ضـــئيلةٍ من المتصفحات الحديثة.

لا أحاول في هذا الكتاب أن أركِّز جهدي على مواصفة معيّنة لتقنيات DOM أو CSS أو HTML؛ فليس الغرض هاهنا هو شرح مواصفة معيّنة، وسيكون ذلك هدفًا كبيرًا (وأرى أنَّه ليس ذا نفعٍ كبيرٍ في كتاب) بعد أخذنا عدد المواصفات المتوافرة حاليًا وتاريخ المتصفحات ومقدار تطبيقها لتلك المواصفات بالحسبان. استعملتُ في طيات فصول الكتاب بعض المحتوى المأخوذ من عدّة مواصفات (Document Object Model (DOM) Level 3 Core Specification، و DOM4، و Document Object Model HTML، و Element Traversal Specification، و Selectors API، و Level 2، و DOM Parsing and Serialization، و HTML 5 Reference، و HTML 5 Specification، و HTML Living Standard، و HTML 5 - A technical specification for Web Developers، و DOM Living Standard)، إلا أنَّ جلَّ محتوى هذا الكتاب مأخوذٌ من ما يستعمله مجتمع المطورين ولا أحاول أن أشرح معيارًا بعينه.

سأشرح بعض المواضيع التي لا تتعلق مباشرةً بميزات DOM، ووضعتُ هذه المواضيع في الكتاب لكي يبني القارئ فهمًا جيدًا لعلاقة DOM مع CSS و JavaScript.

وأهملتُ ذكر بعض التفاصيل التي تتعلق بلغة XML أو XHTML عمدًا، وتجنبتُ شرح الواجهة البرمجية للنماذج والجداول لأحاول إبقاء هذا الكتاب صغيرًا، إلا أنني قد أضيفها إلى الكتاب مستقبلًا في إصدارٍ لاحق.

3. يختلف هذا الكتاب عن بقية الكتاب البرمجية

تَوَجُّهُ هذه السلسلة من الكتب («تعلم jQuery» و «تعلم JavaScript») هو توفير شيفراتٍ صغيرةٍ قابلةٍ للتنفيذ ومعزولةٍ عن بعضها بدلًا من شرح المفاهيم شرحًا كلاميًا مطولًا وبناء برامج كبيرة يصعب فهمها بعد تضخمها. قال أحدهم أنَّ «الكلمات هي أقل مستوى من مستويات التواصل بين البشر» وأوافقه قوله وأجعل من ذلك أساسًا لأسلوبي في صياغة هذه السلسلة. أرى أنَّ من الأفضل شرح المواضيع التقنية بأقل عدد كلمات ممكن، لكن دون الإغفال عن وضع قدرٍ مناسبٍ من الشيفرات مع تعليقاتٍ عليها إذ إنَّ ذلك ضروريٌ ولا بُدَّ منه لإيصال الفكرة.

أسلوب هذا الكتاب يحاول أن يُقدِّم فكرةً معرفةً تعريفًا جيدًا ببلاغةٍ، مدعومةً بشيفرةٍ حقيقية. ولهذا عليك عندما تحاول استيعاب المفاهيم المشروحة أن تُنفِّذ الشيفرة وتتفحصها جيدًا، وبالتالي ستُهيئ نفسك للكلمات المستخدمة لشرح ذاك المفهوم. وحاولتُ أيضًا في هذه السلسلة أن أُقسِّم الأفكار إلى أصغر الأجزاء وراعيتُ شرحها بسياقٍ منفصل كيلا تتداخل المفاهيم مع بعضها بعضًا وتختلط على القارئ.

كل ما سبق يُشير إلى نتيجةٍ وحيدةٍ ألا وهي أنَّ هذا الكتاب ليس كتابًا مطولًا، ولم يأتِ ليشرح مواضيع متفرقة شرحًا واسعًا مستفيضًا.

4. تنسيق الكتاب

قبل أن تبدأ، من المهم أن تفهم طريقة تنسيق الكتاب، رجاءً لا تتخطى هذا القسم لأنه يحتوي على معلوماتٍ مهمةٍ ستساعدك أثناء قراءتِكَ لهذا الكتاب.

رجاءً تفحّص الشيفرات بدقة. يجب أن تنظر إلى الشرح كأمر ثانوي ملحقٌ بالشيفرة. شخصيًا أرى أنَّ الشيفرة تساوي ألف كلمة. لا تقلق إن زاد الشرح حيرتك في البداية، إذ عليك أن تتفحص الشيفرة وأن تقرأ التعليقات مرةً أخرى وتكرِّر هذه العملية إلى أن يصبح المفهوم أو الفكرة الذي أحاول شرحه واضحًا. أرجو أن تصل إلى مرحلةٍ من الخبرة لكيلا تحتاج إلا إلى شيفرةٍ موثقةٍ توثيقًا جيدًا لكي تستوعب أحد المفاهيم البرمجية.

سأستخدمُ الخط العريض في الأمثلة (كما في المثال الآتي) للإشارة إلى الشيفرات والأسطر البرمجية التي تتعلق مباشرةً بالمفهوم الذي نشرحه، وسأستعمل اللون الفضي الفاتح للإشارة إلى التعليقات:

```
<!DOCTYPE html><html lang="en"><body><script>

// هذا تعليقٌ عن الشيفرة لتوضيحها
var foo = 'calling out this part of the code';

</script></body></html>
```

أغلبية أمثلة هذا الكتاب مرتبطة بصفحة خاصة بها في jsFiddle (حيث أضع قبل الشيفرة رابطًا إليها بعنوان «مثال حي»)، حيث يمكنك تعديل وتنفيذ الشيفرة مباشرةً؛ أمثلة jsFiddle تستخدم إضافة Firebug lite-dev لذا ستعمل دالة عرض الناتج (وهي `console.log`) في معظم المتصفحات الحديثة دون مشاكل. أرى أنَّ عليك أن تتعرف على الغرض من الدالة

`console.log` وكيفية استخدامها قبل قراءة هذا الكتاب؛ وهنالك بعض الحالات التي يُسبِّب jsFiddle مشاكل فيها، لذا قررتُ ألّا ضع تجربةً حيةً لتلك الأمثلة.

الفصل الأول:

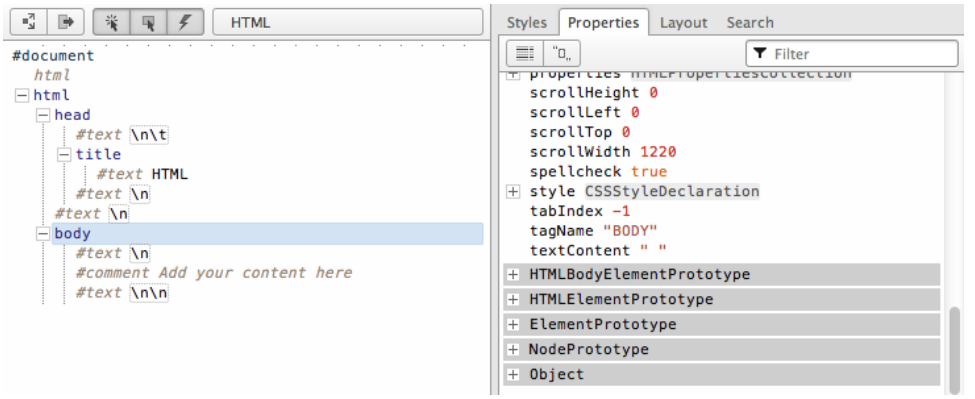
لمحة عن العقد في شجرة DOM

1. DOM هي شجرةٌ تتألف من كائنات JavaScript

عندما تكتب مستند HTML فأنت تُضمِّن عناصر HTML داخل عناصر HTML أخرى، وبفعلك لذلك ستضبط هيكليةً يمكن التعبير عنها كشجرة، وتوضَّح عادةً تلك الهيكلية بصريًا بمحاذاة الوسوم في شيفرات HTML. يُفسِّر المتصفح عند تحميله لمستند HTML هذه الهيكلية ويُنشِئ شجرةً من الكائنات العقدية (node objects) التي تُحاكي الطريقة التي كُتِبَت فيها الشيفرة.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>HTML</title>
</head>
<body>
<!-- أضف المحتوى هنا -->
</body>
</html>
```

ستُنشِئ شيفرة HTML السابقة -عند تفسيرها من المتصفح- مستندًا يحتوي على العقد (nodes) مُهيكَلَةً على شكل شجرة (أي شجرة DOM). هذه الصورة توضِّح بنية الشجرة من مستند HTML السابق باستخدام أدوات المطوِّر في متصفح Opera:



سترى على الجانب الأيسر من الصورة مستند HTML على شكل شجرة، وعلى اليمين سترى كائن JavaScript الذي يُمثِّل العنصر المُحدَّد. فمثلًا، العنصر المُحدَّد <body> ملوّنٌ باللون الأزرق وهو عقدة عنصر (element node) وهو نسخةٌ مبنيةٌ من الواجهة (interface) المسماة `HTMLBodyElement`.

ما عليك أن تفهمه هنا هو أنَّ مستندات HTML تُفسّر من المتصفح وتحوّل إلى بنيةٍ شجريةٍ تتألفُ من كائناتٍ تُمثِّل عقدًا التي بدورها تُمثِّل المستند. الهدف من DOM هو توفير واجهة برمجية لكتابة السكربتات لإجراء عمليات على المستند (مثل الحذف والإضافة والاستبدال والتعديل وربط العناصر مع الأحداث...) باستخدام JavaScript.

ملاحظة

كانت DOM هي الواجهة البرمجية (API) للتعامل مع مستندات XML، إلا أنَّها توسّعت وأصبحت مستخدمةً في مستندات HTML.

2. أنواع العقد

سنورد هنا أشهر أنواع العقد (أي nodeType أو تصنيفات العقد) التي ستواجهك عند تعاملك مع مستندات HTML (لن أذكرها كلها):

- DOCUMENT_NODE (مثلًا: window.document)
- ELEMENT_NODE (مثلًا: <body> أو <a> أو <p> أو <script> أو <style> أو <html> أو <h1> ...إلخ.)
- ATTRIBUTE_NODE (مثلًا: class="funEdges")
- TEXT_NODE (أي المحارف النصية في مستند HTML بما فيها محارف الانتقال إلى سطرٍ جديد والفراغات...)
- DOCUMENT_FRAGMENT_NODE (مثلًا: document.createDocumentFragment())
- DOCUMENT_TYPE_NODE (مثلًا: <!DOCTYPE html>)

ذكرتُ أسماء أنواع العقد السابقة بنفس تنسيق ورودها في المتصفحات كخاصيةٍ (property) تابعةٍ للكائن Node (أي بأحرفٍ كبيرةٍ يُفصَل بين كلماتها بشرطةٍ سفلية _). هذه الخاصيات هي قيمٌ ثابتةٌ (constants) وتُستعمل لتخزين رقمٍ يرتبطُ بنوعٍ معيّن من العقد. ففي المثال الآتي تكون قيمة Node.ELEMENT_NODE مساويةً إلى 1، والقيمة 1 تعني أنَّ العقدة هي عقدة عنصر (element node) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>
// الناتج 1، ويعني أنّ العقدة هي عقدة عنصر
console.log(Node.ELEMENT_NODE)

</script>
</body>
</html>
```

سأعرض في المثال الآتي جميع أنواع العقد وقيمها (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

for(var key in Node){
    console.log(key,' = '+Node[key]);
};

/* ما يلي هو ناتج تنفيذ الشيفرة السابقة
ELEMENT_NODE  = 1
```

```
ATTRIBUTE_NODE  = 2
TEXT_NODE  = 3
CDATA_SECTION_NODE  = 4
ENTITY_REFERENCE_NODE  = 5
ENTITY_NODE  = 6
PROCESSING_INSTRUCTION_NODE  = 7
COMMENT_NODE  = 8
DOCUMENT_NODE  = 9
DOCUMENT_TYPE_NODE  = 10
DOCUMENT_FRAGMENT_NODE  = 11
NOTATION_NODE  = 12
DOCUMENT_POSITION_DISCONNECTED  = 1
DOCUMENT_POSITION_PRECEDING  = 2
DOCUMENT_POSITION_FOLLOWING  = 4
DOCUMENT_POSITION_CONTAINS  = 8
DOCUMENT_POSITION_CONTAINED_BY  = 16
DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC  = 32 */

</script>
</body>
</html>
```

أعطانا المثال السابق قائمةً مفصلةً لجميع أنواع العقد، لكننا لن نشرحها جميعها في هذا الكتاب، وإنما سنناقش قائمةً صغيرةً منها ذكرناها في بداية هذا القسم؛ فمن المرجح أن تتعامل مع تلك الأنواع أثناء كتابة سكربتات تُعدِّل في صفحات HTML.

سأذكر في الجدول الآتي الأسماء المعطية للدوال البانية (constructors) أو الواجهات (interfaces) التي تُهيّئ أغلبية أنواع العقد مع تصنيف nodeType عبر الرقم والاسم. آملُ أن تفهم من الجدول التالي أنَّ قيمة nodeType (مثلًا 1) هي تصنيفٌ رقميٌ الذي يُستخدَم لوصف نوعٍ معيّن من العقد التي تُبنى من دالةٍ بانيةٍ أو واجهةٍ مُحدَّدةٍ في JavaScript. فمثلًا، الواجهة HTMLBodyElement تُمثِّل كائن عقدة ذو النوع 1، وهو التصنيف الذي يوافق ELEMENT_NODE.

الواجهة أو الدالة البانية	نوع العقدة (معادةً من nodeType)
HTML*Element (مثلًا: HTMLBodyElement)	1 (أي ELEMENT_NODE)
Text	3 (أي TEXT_NODE)
HTMLDocument	9 (أي DOCUMENT_NODE)
DocumentFragment	11 (أي DOCUMENT_FRAGMENT_NODE)
DocumentType	10 (أي DOCUMENT_TYPE_NODE)

ملاحظات

- مواصفة DOM تُعَنوِن العقد مثل `Node` و `Element` و `Text` و `Attr` و `HTMLAnchorElement` على أنها «واجهات» (interfaces)، وهي كذلك، لكن أبقِ في ذهنك أنَّ الأسماء نفسها معطيةٌ إلى دالةٍ بانيةٍ (constructor function) في JavaScript التي تبني تلك العقد؛ وستلحظ أنني أشير إلى تلك الواجهات (أقصد `Element` و `Text` و `Attr` و `HTMLAnchorElement`) على أنها كائنات أو دوال بانية، بغض النظر أنَّ المواصفة تُشير إليها على أنها واجهات (أكرِّر أنَّ كلا الأمرين صحيحٌ تمامًا).

- `ATTRIBUTE_NODE` ليست جزءًا من الشجرة لكنها مذكورة لأسبابٍ تاريخية. لن أوفِّر فصلًا عن «عقد الخاصيات» (attribute nodes) وإنما سأناقشها في فصل «عقد العناصر» لأنني أرى أنَّ عقد الخاصيات هي عقدٌ فرعيةٌ من عقد العناصر (element nodes) لكن بدون تمثيل حقيقي في بنية شجرة DOM. أريد أن ألفت انتباهك أنَّ `ATTRIBUTE_NODE` قد أُهمِلَت (depreciated) في DOM 4.

- لم أضع فصلًا في الكتاب لشرح `COMMENT_NODE` لكن اعلم أنَّ التعليقات في مستندات HTML هي عقدٌ من النوع `Comment` وهي ذات طبيعةٍ شبيهةٍ بطبيعة العقد النصية من النوع `Text`.

- ستلحظ أثناء شرحي للعقد في هذا الكتاب أنني أشير أحيانًا إلى عقدةٍ معيّنةٍ بالاسم المعطى لها من الخاصية `nodeType` (مثلًا: `ELEMENT_NODE`)، أفعل ذلك لأجل التوافق مع الاصطلاحات المستخدمة في المواصفات التي توفرها W3C و WHATWG.

3. كائنات العقد الفرعية التي ترث الكائن Node

كل كائن عقدة في شجرة DOM يرث خاصيات ودوال من الكائن Node. واعتمادًا على نوع العقدة في المستند، فقد ترث العقد كائنًا منحدرًا من الكائن Node. سأوضِّح هنا طريقة الوراثة المعتمدة من المتصفحات لواجهات العقد الشائعة:

- الكائن Object > Node > Element > HTMLElement (مثلًا HTML*Element)
- الكائن Object > Node > Attr (أُهمِلَت في DOM 4)
- الكائن Object > Node > CharacterData > Text
- الكائن Object > Node > Document > HTMLDocument
- الكائن Object > Node > DocumentFragment

ليس المهم فقط أن تتذكر أنَّ جميع أنواع العقد ترث الكائن Node فحسب، وإنما عليك أن تدرك أنَّ سلسلة الوراثة قد تطول، فمثلًا جميع عقد HTMLAnchorElement ترث خاصياتها من الكائنات HTMLElement و Element و Node و Object.

ملاحظة

Node هي دالة بانية في JavaScript، وهذا يعني منطقيًا أنَّ Node سترث من Object.prototype كغيرها من الكائنات (نعم، الدوال البانية هي كائنات).

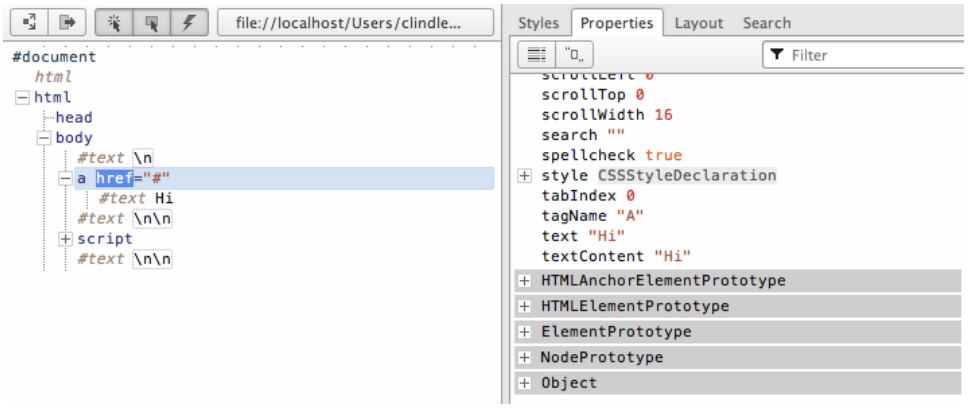
للتأكد أنَّ جميع أنواع العقد ترث خاصياتها من الكائن Node فلنحاول المرور على الكائن Element ونتفحص خاصياته ودواله (بما في ذلك الخاصيات والدوال الموروثة) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

// الحصول على مرجعية لكائن عقدة العنصر
var nodeAnchor = document.querySelector('a');
// إنشاء مصفوفة باسم props لتخزين مفاتيح الخاصيات
// لكائن عقدة العنصر
var props = [];

// المرور على الكائن والحصول على جميع الخاصيات والدوال
// (بما فيها الموروثة)
for(var key in nodeAnchor){
    props.push(key);
}
 // إظهار قائمة مرتبة هجائيًا للخاصيات والدوال
console.log(props.sort());
```

```
</script>
</body>
</html>
```

إذا شغلتَ الشيفرة السابقة في متصفح ويب فسترى قائمةً طويلةً من الخاصيات المتوافرة لكائن عقدة العنصر (element node object). تتضمن هذه القائمة الخاصيات والدوال الموروثة من الكائن `Node` بالإضافة إلى عددٍ لا بأس به من الخاصيات والدوال الموروثة من الكائنات `Element` و `HTMLElement` و `HTMLAnchorElement` و `Node` و `Object`. لا أرمي إلى شرح جميع تلك الخاصيات والدوال الآن لكنني أود أن أشير إلى أنَّ جميع العقد ترث مجموعةً أساسيةً من الخاصيات والدوال من الدالة البانية لها بالإضافة إلى أخذها لخاصيات ودوال أخرى عبر سلسلة prototype.

إذا كنت تحب رؤية سلسلة الوراثة، فيمكنك تفحص مستند HTML السابق في أدوات المطور الخاصة بمتصفح Opera:



لاحظ أنَّ عقدة العنصر <a> ترث من HTMLAnchorElement و HTMLElement و Element و Node و Object وكل تلك الكائنات ظاهرةٌ في قائمة الخاصيات ذات الخلفية الرمادية على الجانب الأيمن من الصورة. توفِّر سلسلة الوراثة (أي سلسلة prototype) عددًا كبيرًا من الدوال والخاصيات المشتركة لجميع أنواع العقد.

ملاحظة	يمكنك الإضافة على شجرة DOM، لكن لا أظن أنَّ من المستحسن تعديل كائنات المضيف (host objects).

4. الخاصيات والدوال المشتركة للعقد

ذكرنا منذ قليل أنَّ جميع كائنات العقد (مثلًا Element و Attr و Text ...إلخ.) ترث الخاصيات والدوال من الكائن Node، والتي تساعدنا في تعديل وتفحص شجرة DOM والتنقل فيها.

وبالإضافة إلى الخاصيات والدوال التي توفرها واجهة Node، فهنالك عددٌ لا بأس به من الخاصيات والدوال التي توفرها الواجهات الفرعية مثل document و HTMLElement.

هذه قائمةٌ بأشهر خاصيات ودوال الكائن Node الموروثة من جميع كائنات العقد، وتتضمن أيضًا الخاصيات الموروثة التي تستعمل للتعامل مع العقد من الواجهات الفرعية.

خاصيات الكائن Node:

- childNodes
- firstChild
- lastChild
- nextSibling
- nodeName
- nodeType
- nodeValue
- parentNode
- previousSibling

دوال الكائن Node:

- appendChild()
- cloneNode()
- compareDocumentPosition()
- contains()
- hasChildNodes()
- insertBefore()
- isEqualNode()
- removeChild()
- replaceChild()

دوال الكائن document:

- document.createElement()
- document.createTextNode()

الخاصيات الموروثة من HTML*Element:

- `lastElementChild`
- `previousElementChild`
- `nextElementChild`
- `children`

دالة موروثة من `Element`:

- `insertAdjacentHTML()`

5. تحديد نوع واسم العقدة

تملك كل عقدة الخاصية `nodeType` و `nodeName` التي ترثها من `Node`. فمثلًا العقد النصية `Text` لها قيمة لخاصية `nodeType` تساوي `3` وقيمة لخاصية `nodeName` تساوي `'#text'`. وكما ذكرتُ سابقًا أنَّ القيمة الرقمية `3` هي شيفرةٌ تمثِّل نوع الكائن الذي تمثِّله العقدة (أي `Node.TEXT_NODE === 3`).

سأفصِّل فيما يلي القيم المُعادة لخاصيتَي `nodeType` و `nodeName` لكائنات العقد التي سنناقشها في هذا الكتاب. أرى أنَّ من الأبسط تَذَكُّرُ القيم الرقمية لأنواع العقد الشائعة لأننا نتعامل عمومًا مع خمسة أنواع منها فقط (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>
```

```
// لأنّ nodeType 10 أو DOCUMENT_TYPE_NODE هذه هي
// Node.DOCUMENT_TYPE_NODE === 10
console.log(
     // الناتج html، جرِّب أيضًا document.doctype لتحصل على
     // <!DOCTYPE html>
    document.doctype.nodeName,
     // الناتج 10، والذي يربط إلى DOCUMENT_TYPE_NODE
    document.doctype.nodeType
);

// لأنّ nodeType 9 أو DOCUMENT_NODE هذه هي
// Node.DOCUMENT_NODE === 9
console.log(
     // الناتج '#document'
    document.nodeName,
     // الناتج 9، والذي يربط إلى DOCUMENT_NODE
    document.nodeType
);

// لأنّ nodeType 11 أو DOCUMENT_FRAGMENT_NODE هذه هي
// Node.DOCUMENT_FRAGMENT_NODE === 11
console.log(
```

```
    // '#document-fragment' الناتج
    document.createDocumentFragment().nodeName,
    // DOCUMENT_FRAGMENT_NODE إلى 11، والذي يربط الناتج
    document.createDocumentFragment().nodeType
);

// لأنّ nodeType 1 أو ELEMENT_NODE هذه هي
// Node.ELEMENT_NODE === 1
console.log(
    // 'A' الناتج
    document.querySelector('a').nodeName,
    // ELEMENT_NODE إلى 1، والذي يربط الناتج
    document.querySelector('a').nodeType
);

// لأنّ nodeType 3 أو TEXT_NODE هذه هي
// Node.TEXT_NODE === 3
console.log(
    // '#text' الناتج
    document.querySelector('a').firstChild.nodeName,
    // TEXT_NODE إلى 3، والذي يربط الناتج
    document.querySelector('a').firstChild.nodeType
);
```

```
</script>
</body>
</html>
```

إن لم تكن الأمور واضحةً لديك، فأحب أن أذكِّر أنَّ أسهل طريقة لتحديد ما هو نوع العقدة هي التحقق من قيمة الخاصية nodeType. سنحقق فيما يلي إن كان العنصر <a> له رقم عقدة مساوٍ للقيمة 1، وإن كان كذلك، فسنعلم أنَّه عقدة من النوع Element لأنَّ Node.ELEMENT_NODE === 1 (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

// هل <a> عقدة من النوع ELEMENT_NODE؟
// الناتج true، لأنّ <a> عقدة عنصر
console.log(document.querySelector('a').nodeType === 1);

// أو يمكننا المقارنة بالخاصية Node.ELEMENT_NODE
// التي تحتوي على القيمة العددية 1
```

```
console.log(document.querySelector('a').nodeType ===
Node.ELEMENT_NODE);

</script>
</body>
</html>
```

تحديد نوع العقدة سيساعدنا كثيرًا عند كتابة السكربتات لأننا سنعرف ما هي الخاصيات والدوال المتاحة إلى العقدة المعنية.

ملاحظة	القيم المُعادة من الخاصية nodeName قد تختلف تبعًا لنوع العقدة. انظر إلى مواصفة DOM 4 للتفاصيل.

6. الحصول على قيمة العقدة

الخاصية `nodeValue` تعيد القيمة `null` لأغلبية أنواع العقد (ما عدا `Text` و `Comment`). واستعمالها مقتصرٌ على استخلاص السلاسل النصية من العقد النصية والتعليقات. سأبيّن ناتج استعمالها على العقد المشروحة في هذا الكتاب في المثال الآتي (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<a href="#">Hi</a>

<script>

// الناتج null لعقد DOCUMENT_TYPE_NODE و DOCUMENT_NODE و
// DOCUMENT_FRAGMENT_NODE و ELEMENT_NODE
console.log(document.doctype.nodeValue);
console.log(document.nodeValue);
console.log(document.createDocumentFragment().nodeValue);
console.log(document.querySelector('a').nodeVale);

// الناتج هو سلسل نصية قيمتها 'Hi'
console.log(document.querySelector('a')
                .firstChild.nodeValue);
</script>
</body>
</html>
```

ملاحظة

يمكن ضبط قيم العقد النصية أو التعليقات عبر إسناد سلاسل نصية جديدة إلى خاصية `nodeValue` (مثلًا: `document.body.firstElementChild.nodeValue = 'hi'`).

7. إنشاء عقد نصية وعناصر باستخدام دوال JavaScript

عندما يُفسِّر المتصفحُ مستندَ HTML ويبني العقد وشجرة DOM بناءً على محتوياته، فسيُنشِئ المتصفح العقد المبدئية عند تحميل مستند HTML. لكن من الممكن إنشاء العقد الخاصة بك باستخدام JavaScript. الدالتان الآتيتان ستسمحان لنا برمجيًا بإنشاء عقد `Element` و `Text` عبر JavaScript:

- `createElement()`
- `createTextNode()`

تتوافر دوال أخرى لكنها غير شائعة الاستخدام (مثل `createAttribute()` و `createComment()`). سأريك في المثال الآتي بساطة إنشاء عقدٍ نصيةٍ وعقدِ عنصر (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

var elementNode = document.createElement('div');
// <div> 1 الناتج هو
// إذ تُشير القيمة 1 إلى أنّ العقدة هي عقدة عنصر
console.log(elementNode, elementNode.nodeType);
```

```
var textNode = document.createTextNode('Hi');
// Text {} 3 الناتج هو
// إذ تُشير القيمة 3 إلى أنّ العقدة هي عقدة نصية
console.log(textNode, textNode.nodeType);
</script>
</body>
</html>
```

ملاحظات

- تقبل الدالة `createElement()` معاملًا وحيدًا هو سلسلةٌ نصيةٌ تُحدِّد ما هو العنصر الذي سيُنشَأ. يجب أن تكون السلسلة النصية مماثلةً للسلسلة المُعادة من الخاصية `tagName` لكائنٍ من النوع `Element`.

- الخاصية `createAttribute()` أصبحت مهملة (deprecated) ولا يجدر بنا استعمالها لإنشاء عقد للخاصيات. يستعمل المطورون عادةً الدوال `getAttribute()` و `setAttribute()` و `removeAttribute()` والتي سأناقشها بالتفصيل في فصل «عقد العناصر» `Element`.

- ستُفصَّل الدالة `createDocumentFragment()` في الفصل الثامن.

- يجب أن تعلم أنَّ الدالة `createComment()` متوافرة لإنشاء عقد للتعليقات، إلا أننا لن نغطيها في هذا الكتاب لكنها متوافرة للمطورين الذين يرون قيمةً في استعمالها.

8. إنشاء وإضافة عقد الكائنات والعقد النصية باستخدام السلاسل النصية في JavaScript

الخاصـــــيات `innerHTML` و `outerHTML` و `textContent` إضـــــافةً إلى الدالـــــة `insertAdjacentHTML()` توفِّر آليةً لإنشاء وإضافة العقد إلى شجرة DOM باستخدام سلاسل نصية عادية في JavaScript.

سنسـتخدم في الشـيفرة الآتيـة الخاصـيات `innerHTML` و `outerHTML` و `textContent` لإنشاء عقد عبر استخدام سلسلة نصية عادية في JavaScript والتي ستُضاف مباشرةً إلى شجرة DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A"></div>
<span id="B"></span>
<div id="C"></div>
<div id="D"></div>
<div id="E"></div>

<script>
```

```
// إنشاء عنصر strong وعقدة نصية وإضافتهما إلى شجرة DOM
document.getElementById('A').innerHTML =
'<strong>Hi</strong>';

// إنشاء عنصر div وعقدة نصية ووضعهما بدلًا من
// <span id="B"></span>
document.getElementById('B').outerHTML = '<div id="B"
class="new">Whats Shaking</div>'

// إنشاء عقدة نصية وتحديث محتوى العنصر div#C
document.getElementById('C').textContent = 'dude';


// استخدام خاصيات غير قياسية (أقصد innerText و outerText)

// إنشاء عقدة نصية وتحديث العنصر div#D
document.getElementById('D').innerText = 'Keep it';

// إنشاء عقدة نصية واستبدال العنصر div#E
// ووضع العقدة النصية بدلًا منه
// (أي أنّ العنصر div#E لم يعد موجودًا)
document.getElementById('E').outerText = 'real!';

console.log(document.body.innerHTML);
```

```
/* الناتج
<div id="A"><strong>Hi</strong></div>
<div id="B" class="new">Whats Shaking</div>
<span id="C">dude</span>
<div id="D">Keep it</div>
real!
*/

</script>
</body>
</html>
```

الدالة `insertAdjacentHTML()` التي تعمل مع عقد `Element` فقط أكثر دقةً من الطرائق سابقة الذكر. فسيمكن باستخدام هذه الدالة إضافة العقد قبل وسم البداية أو بعد وسم البداية أو قبل وسم النهاية أو بعد وسم النهاية. سأركِّب جملةً في المثال الآتي باستخدام الدالة `insertAdjacentHTML()` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body><i id="elm">how</i>
<script>
var elm = document.getElementById('elm');
```

```
elm.insertAdjacentHTML('beforebegin', '<span>Hey-</span>');
elm.insertAdjacentHTML('afterbegin', '<span>dude-</span>');
elm.insertAdjacentHTML('beforeend', '<span>-are</span>');
elm.insertAdjacentHTML('afterend', '<span>-you?</span>');

console.log(document.body.innerHTML);
/* الناتج
<span>Hey-</span><i
id="A"><span>dude-</span>how<span>-are</span></i><span>-you?
</span>
*/
</script>
</body>
</html>
```

ملاحظات

- الخاصية `innerHTML` ستحوّل عناصر HTML الموجودة ضمن السلسلة النصية إلى عقد DOM بينما يمكن استعمال `textContent` لإنشاء العقد النصية فقط. إذا مررتَ سلسلةً نصيةً تحتوي على عناصر HTML إلى `textContent` فستظهر تلك العناصر كنصٍ عادي ولن تُفسَّر.

- يمكن استخدام الدالة `document.write()` لإنشاء وإضافة العقد إلى شجرة DOM معًا. لكنها لم تعد تستعمل في هذه الفترة إلا إن كانت ضروريةً لإنجاز مهام معيّنة. المفهوم الأساسي وراء الدالة `write()` هو إخراج القيم

المُمرَّرة إليها إلى الصفحة أثناء تحميل وتفسير المستند. يجب أن تدرك أنَّ الدالة `write()` ستؤدي إلى إيقاف تفسير مستند HTML مؤقتًا إلى أن ينتهي تنفيذها.

- يؤدي استخدام الخاصية `innerHTML` إلى استدعاء مُفسِّر ثقيل جدًا لشيفرات HTML، في حين أنَّ إنشاء العقد النصية لا يتطلب إلا قدرًا ضئيلًا من الموارد، لذا اقتصد في استخدام `innerHTML` وأخواتها.

- خيارا الدالة `insertAdjacentHTML` اللذان يضيفان العقدة قبل وسم البداية (`beforebegin`) وبعد وسم النهاية (`afterend`) سيعملان إن كانت العقدة لها عنصرٌ أب (parent element) فقط.

- لم تكن الخاصية `outerHTML` مدعومةً دعمًا أصليًا في متصفح Firefox حتى الإصدار 11 منه. لكن يتوفر حلٌ التفافيٌ لهذه المشكلة.

- الخاصية `textContent` تستطيع الحصول على محتوى جميع العناصر بما في ذلك `<script>` و `<style>`، لكن الخاصية `innerText` لا تستطيع فعل ذلك.

- الخاصية `innerText` تأخذ بالحسبان تنسيق العنصر ولن تُعيد المحتوى النصي للعناصر المخفية، بينما ستفعل ذلك الخاصية `textContent`.

9. استخلاص أجزاء من شجرة DOM كسلاسل نصية في JavaScript

يمكننا استخدام نفس الخاصيات (`innerHTML` و `outerHTML` و `textContent`) التي استخدمناها لإنشاء وإضافة العقد إلى شجرة DOM لاستخلاص أجزاء من شجرة DOM (أو حتى شجرة DOM كلها) على هيئة سلسلةٍ نصيةٍ في JavaScript. سأستخدمُ في المثال الآتي الخاصيات السابقة لإعادة سلسلة نصية تحتوي على قيم نصية وشيفرات HTML من المستند (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A"><i>Hi</i></div>
<div id="B">Dude<strong> !</strong></div>

<script>

// الناتج '<i>Hi</i>'
console.log(document.getElementById('A').innerHTML);

// الناتج <div id="A">Hi</div>
console.log(document.getElementById('A').outerHTML);
```

```
// لاحظ إعادة المحتوى النصي كله حتى لو كان موجودًا
// (<strong> !</strong> أقصد) في عقدة عنصر فرعية
// 'Dude !' الناتج
console.log(document.getElementById('B').textContent);

// (outerText و innerText أقصد) استخدام خاصيات غير قياسية

// 'Dude !' الناتج
console.log(document.getElementById('B').innerText);
// 'Dude !' الناتج
console.log(document.getElementById('B').outerText);

</script>
</body>
</html>
```

ملاحظة

عندما نحصل على قيمة من الخاصية `textContent` فستُعاد جميع العقد النصية الموجودة في العقدة المعنية، فمثلًا لو حاولنا الوصول إلى الخاصية `document.body.textContent` (لا أنصحك بفعل ذلك في المواقع الإنتاجية) فستُعيد محتوى جميع العقد النصية الموجودة في العنصر body، وليس أوّل عقدة نصية فقط.

10. إضافة كائنات العقد إلى شجرة DOM باستخدام appendChild() و insertBefore()

تسمح لنا الدالتان (الموروثتان من Node) `appendChild()` و `insertBefore()` بإضافة كائنات العقد المُنشَأة باستخدام JavaScript إلى شجرة DOM.

الدالة `appendChild()` ستُضيف العقدة بعد نهاية العقد الأبناء للعنصر المُحدَّد، وفي حال عدم وجود عقد أبناء فستُضاف العقدة كأوّل عقدة ابن للعنصر المحدد. سنُنشِئ في المثال الآتي عقدة عنصر (`<strong>`) وعقدةً نصيةً (`Dude`)، ثم سنُحدِّد العنصر `<p>` من شجرة DOM لكي نُضيف إليه العنصر `<strong>` الذي أنشأناه باستعمال الدالة `appendChild()`. لاحظ أنَّ العنصر `<strong>` مغلّفٌ داخل العنصر `<p>` وأُضيف كآخر ابن له. الخطوة التالية هي تحديد العنصر `<strong>` وإضافة العقدة النصية `'Dude'` إليه (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<p>Hi</p>
<script>

// إنشاء عقدة عنصر فارغة وعقدةً نصيةً
var elementNode = document.createElement('strong');
var textNode = document.createTextNode(' Dude');
```

```
// إضافة العقدتين السابقتين إلى شجرة DOM
document.querySelector('p').appendChild(elementNode);
document.querySelector('strong').appendChild(textNode);

// الناتج <p>Hi<strong> Dude</strong></p>
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

وعندما يكون التحكم في مكان الإضافة ضروريًا عوضًا عن إضافة العقد بعد نهاية العقد الأبناء، فحينئذٍ نستعمل `insertBefore()`. إذ سأضيف العنصر `<li>` في المثال الآتي قبل أوّل عقدة تابعة للعنصر `<ul>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<ul>
    <li>2</li>
    <li>3</li>
</ul>
```

```
<script>

// إنشاء عقدة نصية وعقدة عنصر li
// وإضافة العقدة النصية إلى العنصر li
var text1 = document.createTextNode('1');
var li = document.createElement('li');
li.appendChild(text1);

// تحديد العنصر ul في المستند
var ul = document.querySelector('ul');



/*
 إضافة العنصر li الذي أنشأناه إلى شجرة DOM، لاحظ أنني حددتُ
 العنصر <ul> ومررتُ المرجعية ul.firstChild إلى <li>2</li>
*/
ul.insertBefore(li,ul.firstChild);

console.log(document.body.innerHTML);
/* الناتج
<ul>
<li>1</li>
<li>2</li>
<li>3</li>
```

```
</ul>
*/

</script>
</body>
</html>
```

الدالة `insertBefore()` تتطلب معاملين (parameters)، أولهما هو العقدة التي ستُضاف، وثانيهما هو مرجعٌ إلى العقدة الموجودة في المستند التي تريد إضافة العقدة قبلها.

ملاحظات

- إذا لم تُمرِّر معاملًا ثانيًا إلى الدالة `insertBefore()` فستسلك نفس سلوك الدالة `appendChild()`.
- لدينا دوالٌ أخرى (مثل `prepend()` و `append()` و `before()` و `after()`) لنتطلع إليها في DOM 4.

11. حذف واستبدال العقد باستخدام removeChild() و replaceChild()

تتألف عملية حذف عقدة من شجرة DOM من عدّة خطوات. إذ عليك أولًا تحديد العقدة التي تريد حذفها، ثم تحصل على وصولٍ إلى العنصر الأب لها عبر الخاصية `parentNode`، ثم عليك تنفيذ الدالة `removeChild()` على العقدة الأب مُمرِّرًا إليها مرجعيةً إلى العقدة التي تود حذفها.

سأوضِّح ذلك في المثال الآتي على عقدة عنصر وعقدةٍ نصية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A">Hi</div>
<div id="B">Dude</div>

<script>

// حذف عقدة العنصر
var divA = document.getElementById('A');
divA.parentNode.removeChild(divA);

// حذف العقدة النصية
var divB = document.getElementById('B').firstChild;
divB.parentNode.removeChild(divB);

// إظهار ما تغيّر في شجرة DOM
// إذ سيظهر فيها العنصر div#B الفارغ فقط
console.log(document.body.innerHTML);

</script>
```

```
</body>
</html>
```

لا تختلف آلية استبدال عنصر عن آلية حذفه كثيرًا. سأستعمل في المثال الآتي نفس بنية مستند HTML في المثال السابق لكنني هذه المرة سأستخدم الدالة `replaceChild()` لتحديث العقد بدلًا من حذفها (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div id="A">Hi</div>
<div id="B">Dude</div>
<script>

// استبدال عقدة العنصر
var divA = document.getElementById('A');
var newSpan = document.createElement('span');
newSpan.textContent = 'Howdy';
divA.parentNode.replaceChild(newSpan,divA);

// استبدال العقدة النصية
var divB = document.getElementById('B').firstChild;
var newText = document.createTextNode('buddy');
```

```
divB.parentNode.replaceChild(newText, divB);

// إظهار ما تغيّر في شجرة DOM
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

ملاحظات

- يجدر بالذكر أنَّ توفير سلسلة نصية فارغة إلى الخاصيات `innerHTML` أو `outerHTML` أو `textContent` قد يكون أسهل وأسرع مما سبق، إلا أنَّ ذلك يعتمد على العقد التي تحاول حذفها أو استبدالها. احذر من حدوث تسريب في الذاكرة (memory leaks) في المتصفح.

- تُعيد الدالتان `replaceChild()` و `removeChild()` العقدة التي نُفِّذَت عليها، فهي لم تختفِ بمجرد استبدالها أو حذفها، وإنما لم تعد موجودةً في النسخة الحالية الحية (live) من المستند، لكن ما يزال بإمكانك الإشارة إليها في الذاكرة.

- هنالك دوالٌ إضافيةٌ (مثل `replace()` و `remove()`) لنتطلع إليها في DOM 4.

12. نسخ العقد باستخدام cloneNode()

من الممكن عند استخدام الدالة `cloneNode()` أن نستنسخ عقدةً وحيدةً، أو عقدةً مع جميع العقد الأبناء التابعة لها.

سأنسخ في هذه الشيفرة العنصر `<ul>` فقط (أقصد `HTMLULListElement`) والذي بعد نسخه سنستطيع التعامل معه كغيره من المرجعيات إلى العقد (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<ul>
  <li>Hi</li>
  <li>there</li>
</ul>
<script>

var cloneUL = document.querySelector('ul').cloneNode();

// HTMLULListElement() الناتج
console.log(cloneUL.constructor);
// الناتج هو سلسلةٌ نصيةٌ فارغةٌ لعدم نسخ أي شيء
// عدا العنصر ul
console.log(cloneUL.innerHTML);
```

```
</script>
</body>
</html>
```

أما لنسخ عقدة وجميع العقد الأبناء فيها، فمرِّر معاملًا قيمته `true` إلى الدالة `cloneNode()`. سأستعمل الدالة `cloneNode()` مرةً أخرى، لكنها ستنسخ في هذه المرة جميع العقد الأبناء أيضًا (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<ul>
  <li>Hi</li>
  <li>there</li>
</ul>
<script>

var cloneUL = document.querySelector('ul').cloneNode(true);

// HTMLUListElement() الناتج
console.log(cloneUL.constructor);
// <li>Hi</li><li>there</li> الناتج
console.log(cloneUL.innerHTML);
```

```
</script>
</body>
</html>
```

ملاحظات

- عند نسخ عقدة عنصر (أي `Element`) فستنسخ جميع خاصياتها وقيم تلك الخاصيات. لكن انتبه أنَّه لن يُنسَخ إلا الخاصيات فقط! إذ سيضيع أيُّ شيءٍ آخر تضيفه إلى عقدة DOM (مثل دوال التعامل مع الأحداث) عند نسخها.

- ربما تظن أنَّ نسخ العقدة مع جميع أبنائها (`cloneNode(true)` سيُعيد قائمة `NodeList` لكنه لن يفعل ذلك.

- قد يؤدي استخدام `cloneNode()` إلى تكرر مُعرِّف `id` العنصر في المستند.

13. فهم مجموعات العقد (NodeList و HTMLcollection)

عند تحديد مجموعات من العقد من شجرة DOM (سنشرح ذلك في الفصل الثالث «عقد العناصر») أو الوصول إلى مجموعة معرّفة مسبقًا من العقد، فستكون تلك العقد موجودةً في `NodeList` (مثلًا `document.querySelectorAll('*')`) أو `HTMLcollection` (مثلًا `document.scripts`). تلك المجموعات (collections) تشبه المصفوفات (لكنها ليس مصفوفات حقيقية) ولها الخصائص الآتية:

- تكون المجموعة إما حيةً (live) أو ثابتةً (static). هذا يعني أنَّ العقد الموجودة في المجموعة إما أن تكون جزءًا من المستند الحي الحالي أو كانت موجودةً في نسخةٍ سابقةٍ من المستند.
- تُرتَّب العقد افتراضيًا داخل المجموعة حسب ترتيبها في الشجرة، وهذا يعني أنَّ ترتيب يكون خطيًّا من «ساق» الشجرة إلى «أغصانها».
- لدى المجموعات خاصية `length` التي تخزِّن عدد العناصر في القائمة.

14. الحصول على قائمة لجميع العقد الأبناء

استخدام الخاصية `childNodes` سيؤدي إلى توليد قائمة شبيهة بالمصفوفات (أقصد `NodeList`) لجميع العقد الأبناء المباشرة لها (immediate child). سأحدِّد في المثال الآتي العنصر `<ul>` ثم سأستخدمه لإنشاء قائمة لجميع العقد الأبناء الموجودة مباشرةً ضمن العنصر `<ul>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<ul>
  <li>Hi</li>
  <li>there</li>
</ul>
```

```
<script>

var ulElementChildNodes =
document.querySelector('ul').childNodes;

// الناتج هو قائمة شبيهة بالمصفوفات فيها جميع
// العقد الموجودة مباشرةً في ul
console.log(ulElementChildNodes);



/*
استدعاء الدالة forEach كما لو أنها دالة تابعة للكائن
NodeList لكي نتمكن من الدوران على عناصر القائمة. فعلنا ذلك
لأنّ الكائن NodeList شبيهٌ بالمصفوفات إلا أنه لا يرث الكائن
Array مباشرةً
*/
Array.prototype.forEach.call(ulElementChildNodes,function(ite
m){
   // إظهار كل عنصر موجود في «المصفوفة»
   console.log(item);
});
</script>
</body>
</html>
```

ملاحظات

- القائمة من النوع `NodeList` المُعادة من الخاصية `childNodes` ستحتوي على العقد الموجودة مباشرةً في العقدة المعنية (أي لن تظهر العقد المتشعبة).

- لاحظ أنَّ الخاصية `childNodes` لا تُعيد عقد العناصر (من النوع `Element`) فقط، وإنما ستُعيد أيضًا جميع الأنواع الأخرى من العقد (مثل العقد النصية `Text` والتعليقات `Comment`).

- أضيف دعم `[].forEach` في الإصدار الخامس من ECMAScript.

15. تحويل كائن NodeList أو HTMLCollection إلى مصفوفة Array

الكائنان `NodeList` و `HTMLCollection` هما كائنان شبيهان بالمصفوفات إلا أنهما ليسا مصفوفاتٍ حقيقة في JavaScript لذا لن يرثا الدوال الخاصة بها. سأريك ذلك برمجيًا باستخدام الدالة `isArray()` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<a href="#"></a>
<script>

// الناتج false، لأنّ HTMLCollection ليست مصفوفةً
```

```
console.log(Array.isArray(document.links));

// الناتج false، لأنّ NodeList ليست مصفوفةً
console.log(Array.isArray(document.querySelectorAll('a')));
</script>
</body>
</html>
```

ملاحظة

الدالة `Array.isArray` أصبح متاحةً منذ الإصدار الخامس من ECMAScript (أي ES5).

سنستفيد كثيرًا من تحويل كائنات `NodeList` أو `HTMLCollection` إلى مصفوفات JavaScript حقيقية. أولًا، سنتمكن من إنشاء نسخة من القائمة دون أن ترتبط بشجرة DOM الحية في المستند بعد الأخذ بالحسبان أنَّ `NodeList` و `HTMLCollection` هما قائمتان حيتان (live list). وثانيًا، تحويل قائمة إلى مصفوفة JavaScript سيعطينا وصولًا إلى الدوال الموجودة في الكائن `Array` (مثلًا: `forEach` و `pop` و `map` و `reduce` ...إلخ.).

لتحويل قائمة شبيهة بالمصفوفات إلى مصفوفة JavaScript، فمرِّر القائمة إلى الدالة `call()` أو `apply()`، والتي ستستدعي دالةً تُعيد مصفوفةً حقيقةً غيرَ معدلةٍ سأستخدم في المثال الآتي الدالة `slice()`. التي لن تُقسِّم أيّ شيء وإنما سأستعملها لتحويل قائمة إلى مصفوفة من النوع `Array` (ذلك لأنَّ القيمة المُعادة من الدالة `slice()` هي مصفوفة) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<a href="#"></a>
<script>

// true الناتج
console.log(Array.isArray(Array.prototype.slice.call(document
.links)));
// true الناتج
console.log(Array.isArray(Array.prototype.slice.call(document
.querySelectorAll('a'))));
</script>
</body>
</html>
```

ملاحظة

في الإصدار السادس من ECMAScript، هنالك دالة باسم `Array.from` التي تحوِّل وسيطًا وحيدًا هو قائمةٌ أو كائنٌ شبيه بالمصفوفات مُمرَّرًا إليها كمعامل (مثل `arguments` أو `NodeList` أو `DOMTokenList` [المستعمل من `classList`] أو `NamedNodeMap` [المستعمل من الخاصية `attributes`]) إلى مصفوفة `Array()` ثم تُعيد تلك المصفوفة.

16. التنقل في عقد شجرة DOM

من الممكن عبر استعمالنا لمرجع عقدةٍ ما (مثلًا `document.querySelector('ul')`) أن نحصل على مرجعٍ لعقدةٍ مختلفةٍ عبر التنقل في شجرة DOM باستخدام الخاصيات الآتية:

- `parentNode`
- `nextSibling`
- `firstChild`
- `previousSibling`
- `lastChild`

سنستعمل خاصيات الكائن `Node` في المثال الآتي لتبيان آلية التنقل في شجرة DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body><ul><!-- comment -->
<li id="A"></li>
<li id="B"></li>
<!-- comment -->
</ul>

<script>

// تخزين قيمة المرجعية إلى عقدة ul مؤقتًا
var ul = document.querySelector('ul');
```

```
// ما هي العقدة الأب (parentNode) للعنصر ul؟
// الناتج body
console.log(ul.parentNode.nodeName);

// ما هو أوّل ابن (firstChild) للعنصر ul؟
// الناتج comment
console.log(ul.firstChild.nodeName);

// ما هو آخر ابن (lastChild) للعنصر ul؟
// الناتج هو text وليس comment
// والسبب هو وجود سطر إضافي بعد التعليق
console.log(ul.lastChild.nodeName);

// ما هي العقدة «الأخ» التالية (nextSibling) لأوّل عنصر li؟
// الناتج text
console.log(ul.querySelector('#A').nextSibling.nodeName);

// ما هي العقدة «الأخ» السابقة (previousSibling)
// لآخر عنصر li؟
// الناتج text
console.log(ul.querySelector('#B').previousSibling.nodeName);
</script>
</body>
</html>
```

إذا تعاملتَ كثيرًا مع DOM فيجب ألّا يفاجئك أنَّ التنقل في DOM لا يتضمن التنقل بين عقد العناصر فحسب وإنما يتضمن العقد النصية والتعليقات. أتوقع أنَّ آخر مثال قد وضَّح هذه الفكرة. يجدر بالذكر أنَّنا نستطيع استخدام الخاصيات الآتية للتنقل في شجرة DOM مع تجاهل العقد النصية والتعليقات:

- `firstElementChild`
- `lastElementChild`
- `nextElementChild`
- `previousElementChild`
- `children`

ملاحظة

لم نذكر الخاصية `childElementCount` لكنها موجودة وتستعمل لحساب عدد العناصر الأبناء لعقدةٍ ما.

انظر مرةً أخرى إلى المثال السابق لكن مع استعمال دوال التنقل بين عقد العناصر (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body><ul><!-- comment -->
<li id="A"></li>
<li id="B"></li>
<!-- comment -->
```

```
</ul>

<script>

// تخزين قيمة المرجعية إلى عقدة ul مؤقتًا
var ul = document.querySelector('ul');

// ما هو أوّل ابن (firstElementChild) للعنصر ul؟
// الناتج li
console.log(ul.firstElementChild.nodeName);

// ما هو آخر ابن (lastElementChild) للعنصر ul؟
// الناتج li
console.log(ul.lastElementChild.nodeName);

// ما هو العنصر «الأخ» التالي (nextElementSibling)
// لأوّل عنصر li؟
// الناتج li
console.log(ul.querySelector('#A').nextElementSibling.nodeNam
e);

// ما هو العنصر «الأخ» السابق (previousElementSibling)
// لآخر عنصر li؟
// الناتج li
```

```
console.log(ul.querySelector('#B').previousElementSibling.nod
eName);

// ما هي العناصر الموجودة كعقد أبناء للعنصر ul؟
// الناتج هو HTMLCollection،
// يتضمن جميع العقد الأبناء بما فيها العقد النصية
console.log(ul.children);

</script>
</body>
</html>
```

17. التحقق من موضع عقدة في شجرة DOM باستخدام contains() و compareDocumentPosition()

من الممكن معرفة إن كانت عقدةٌ نصيةٌ موجودةً داخل عقدةٍ أخرى باستخدام الدالة التابعة للكائن Node ذات الاسم contains(). سأحاول في الشيفرة الآتية معرفة إن كان العنصر <body> موجودًا ضمن <html lang="en"> (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<script>

// هل العنصر <body> موجودٌ ضمن <html lang="en">؟
var inside =
document.querySelector('html').contains(document.querySelecto
r('body'));

// النا تج true
console.log(inside);

</script>
</body>
</html>
```

إذا أردت الحصول على معلوماتٍ أدق عن مكان العقدة في شجرة DOM وذلك وفقًا لموقعها نسبةً إلى العقد التي حولها، فيمكنك استعمال الدالة الموجودة في كائن Node ذات الاسم compareDocumentPosition()، وهي دالةٌ تعطيك إمكانية طلب معلومات حول مكان العقدة المُحدَّدة نسبةً إلى العقدة التي مُرِّرَت إلى تلك الدالة. المعلومات التي ستحصل عليها هي رقمٌ يرتبط بالمعلومات التالية:

الرقم المُعاد من الدالة `compareDocumentPosition()`	شرح قيمة ذاك الرقم
0	العنصران متماثلان تمامًا.
1	`DOCUMENT_POSITION_DISCONNECTED` أي أنَّ العقدتين ليستا في نفس المستند.
2	`DOCUMENT_POSITION_PRECEDING` أي أنَّ مكان العقدة المُمرَّرة إلى الدالة في شجرة DOM يسبق العقدة المُحدَّدة.
3	`DOCUMENT_POSITION_FOLLOWING` أي أنَّ مكان العقدة المُمرَّرة يلي العقدة المحدَّدة.
8	`DOCUMENT_POSITION_CONTAINS` أي أنَّ العقدة المُمرَّرة هي عنصرٌ أب (مباشر أو غير مباشر) للعقدة المُحدَّدة.
16، 10	`DOCUMENT_POSITION_CONTAINED_BY` (توافق القيمة 16، أو 10 بالنظام الست عشري). أي أنَّ العقدة المُمرَّرة هي من سلالة (ابن مباشر أو غير مباشر) العقدة المُحدَّدة.

ملاحظات

- الدالة `contains()` ستُعيد `ture` إذا كانت العقدتين -المُحدَّدة والمُمرَّرة إليها- متماثلتين.

- قد تربكك القيمة المُعادة من الدالة `compareDocumentPosition()` في بعض الأحيان لاحتمال تعقيد علاقة عقدة بأخرى. فمثلًا لو كانت العقدة تحتوي (16) على عقدةٍ أخرى وتسبقها (4) فالقيمة المعادة من الدالة `compareDocumentPosition()` هي 20.

18. كيفية معرفة إن كانت العقدتان متماثلتين

وفقًا لمواصفة DOM 3، تكون العقدتان متماثلتين إذا تحققت الشروط الآتية جميعها:

- كانت العقدتان من نفس النوع.
- كانت قيم الخاصيات `nodeName` و `localName` و `namespaceURI` و `prefix` و `nodeValue` متساويةً. أي أنها لا تساوي `null` ولها نفس الطول والمحارف تمامًا.
- الخاصيتان `attributes` متماثلتان في كلا العقدتين، أي أن قيمتهما لا تساوي `null`، ولهما نفس الطول...
- الخاصيتان `childNodes` متساويتان، أي أنها لا تساوي `null` ولهما نفس الطول، وتحتوي على نفس العقد في نفس الترتيب.

يمكن معرفة إن كانت عقدةٌ مساويةً لعقدةٍ أخرى باستدعاء الدالة `isEqualNode()`. على

أولاهما وتمرير الأخرى كمعامل. سأبيّن طريقة عمل الدالة السابقة في هذا المثال بتنفيذها على عقدتين متماثلتين وعقدتين مختلفتين (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<input type="text">
<input type="text">

<textarea>foo</textarea>
<textarea>bar</textarea>
<script>

// الناتج true وذلك لأنّ العقدتين متماثلتين تمامًا
var input = document.querySelectorAll('input');
console.log(input[0].isEqualNode(input[1]));
// الناتج false وذلك لأنّ العقدة النصية مختلفة
var textarea = document.querySelectorAll('textarea');
console.log(textarea[0].isEqualNode(textarea[1]));

</script>
</body>
</html>
```

ملاحظة

إن لم تكن مهتمًا بمعرفة إن كانت عقدتان متماثلتين وإنما أردتَ معرفة إن كانتا تُشيران إلى نفس العقدة، فيمكنك حينئذٍ التحقق من ذلك عبر معامل المطابقة === (مثلًا `document.body === document.body`). وهذا سيخبرنا أنَّهما يشيران إلى نفس العقدة تمامًا ولن يخبرنا إن كانتا متماثلتين أم لا.

الفصل الثاني:

عقدة المستند

2

1. لمحة عن العقدة document

عند تهيئة الدالة البانية HTMLDocument (التي ترث من الكائن document) فستُمثِّل عقدةً خاصة من النوع DOCUMENT_NODE (أي windows.document) في شجرة DOM. وللتأكد من ذلك، فسنحاول معرفة ما هي الدالة البانية المستعملة لإنشاء كائن العقدة document (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

// function HTMLDocument() { [native code] } الناتج هو
console.log(window.document.constructor);
// 9 الناتج
// DOCUMENT_NODE وهو القيمة الرقمية المرتبطة بالنوع
console.log(window.document.nodeType);

</script>
</body>
</html>
```

النتيجة المستخلصة من الشيفرة السابقة هي أنَّ الدالة البانية HTMLDocument ستُنشِئ الكائن window.document الذي هو كائن من النوع DOCUMENT_NODE.

ملاحظة

ستُهيّأ الدالتان البانيتان `Document` و `HTMLDocument` عادةً من المتصفح عندما تُحمِّل مستند HTML. لكن عبر استخدام الدالة `document.implementation.createHTMLDocument()` سيصبح من الممكن إنشاء مستند HTML مستقل عن المستند المُنشَأ في المتصفح. وبالإضافة إلى الدالة `createHTMLDocument()`، هنالك إمكانيةٌ لإنشاء كائن `document` الذي سيُهيّأ ليصبح مستند HTML وذلك عبر الدالة `createDocument()`. يرتبط الاستخدام الاعتيادي لهذه الدوال بتوفير مستند HTML إلى عنصر `iframe` برمجيًا.

2. خاصيات ودوال الكائن HTMLDocument (بما فيها الموروثة)

للحصول على معلومات دقيقة فيما يخص الخاصيات والدوال المتوافرة لكائن العقدة `HTMLDocument`، فالأفضل تجاهل المواصفة وسؤال المتصفح عنها. تفحّص المصفوفات المُنشَأة في الشيفرة الآتية التي تفصِّل ما هي الخاصيات والدوال المتوافرة لكائن من النوع `HTMLDocument` (أقصد الكائن `window.document`) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>
```

```
// document الخاصيات التابعة مباشرةً للكائن
console.log(Object.keys(document).sort());

// (بما فيها الموروثة) document الخاصيات التابعة للكائن
var documentPropertiesIncludeInherited = [];
for(var p in document){
    documentPropertiesIncludeInherited.push(p);
}
console.log(documentPropertiesIncludeInherited.sort());

// document خاصيات الكائن الموروثة فقط
var documentPropertiesOnlyInherited = [];
for(var p in document){
    if(
          !document.hasOwnProperty(p))
{documentPropertiesOnlyInherited.push(p);
    }
}
console.log(documentPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

الخاصيات المتوافرة لهذا الكائن كثيرة، حتى لو لم نأخذ بالحسبان الخاصيات الموروثة. هذه قائمةٌ مختصرةٌ بالخاصيات والدوال التي تلفت النظر والتي سنشرحها في طيّات هذا الفصل:

- `doctype`
- `documentElement`
- `implementation.*`
- `activeElement`
- `body`
- `head`
- `title`
- `lastModified`
- `referrer`
- `URL`
- `defaultview`
- `compatMode`
- `ownerDocument`
- `hasFocus()`

ملاحظة

يُستخدَم الكائن `HTMLDocument` عادةً للوصول إلى قدرٍ كبيرٍ من الخاصيات والدوال (الموروثة في الأعمّ الأغلب) للتعامل مع DOM (مثلًا: `document.querySelectorAll()`). سترى أنَّ كثيرًا من هذه الخاصيات لن تُشرح في هذا الفصل وإنما سنخصص لها مكانًا في فصولٍ أخرى مناسبة في هذا الكتاب.

3. الحصول على معلومات عامة عن مستند HTML

يُوفِّر الكائن `document` وصولًا إلى بعض المعلومات العامة عن مستند HTML (أو DOM) المُحمّل في المتصفح. سأستخدم في الشيفرة الآتية الخاصيات `document.title` و `document.URL` و `document.referrer` و `document.lastModified`

و `document.compatMode` للحصول على بعض المعلومات العامة عن المستند. يجب أن تكون القيم المعادة من الخاصيات السابقة منطقيةً بالنسبة لك بعد النظر والتفكّر بأسماء الخاصيات التي تُعيد تلك القيم (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

var d = document;
console.log('title = ' +d.title);
console.log('url = ' +d.URL);
console.log('referrer = ' +d.referrer);
console.log('lastModified = ' +d.lastModified);

// الناتج إنما BackCompat (نمط التجاوزات [Quirks Mode])
// أو CSS1Compat (في النمط القياسي)
console.log('compatibility mode = ' +d.compatMode);

</script>
</body>
</html>
```

4. العقد الأبناء لعقد document

يمكن أن تحتوي العقد من النوع Document على كائن DocumentType وحيد وكائن عقدة عنصر Element وحيد. يجب ألّا يفاجئك ذلك لأنَّ مستندات HTML تحتوي عادةٍ على نوع مستند وحيد (نوع المستند هو doctype مثل <!DOCTYPE html>) وعنصر وحيد (مثلًا: <html lang="en">). وبالتالي إذا سألت عن أبناء الكائن Document (مثلًا: document.childNodes) فستحصل على مصفوفة تحتوي على الأقل على نوع المستند (doctype) وعلى عنصر <html>. الشيفرة الآتية تُظهِر نوع العقدة window.document (أي Document) مع العقد الأبناء (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

// doctype/DTD هذا هو
// DOCUMENT_TYPE_NODE الناتج 10، وهو الرقم الذي يرتبط بنوع
console.log(document.childNodes[0].nodeType);

// <html> هذا عنصر
// ELEMENT_TYPE_NODE الناتج 1، وهو الرقم الذي يرتبط بنوع
console.log(document.childNodes[1].nodeType);
```

```
</script>
</body>
</html>
```

ملاحظة

لا تخلط بين الكائن `window.document` المُنشَأ من الدالة البانية `HTMLDocument` مع الكائن `Document`. تذكّر أنَّ `window.document` هو نقطة الانطلاق التي تبدأ منها شجرة DOM. وهذا هو السبب احتواء الخاصية `document.childNodes` على العقد الأبناء.

إذا أُنشِئت عقدةُ تعليقٍ (التي لن نشرحها في هذا الكتاب) خارج العنصر `<html lang="en">` فستصبح عقدة ابن للكائن `window.document`؛ لكن وجود عقد للتعليقات خارج العنصر `<html>` سيؤدي إلى حدوث علل في متصفح IE وهو خرقٌ لمواصفة DOM.

5. يوفِّر الكائن document اختصاراتٍ إلى <!DOCTYPE> و <html lang="en"> و <head> و <body>

استخدم الخاصيات المذكورة أدناه للحصول على مرجعيةٍ إلى العقد الآتية:

- الخاصية `document.doctype` تُشير إلى `<!DOCTYPE>`
- الخاصية `document.documentElement` تُشير إلى `<html lang="en">`

- الخاصية document.head تُشير إلى <head>
- الخاصية document.body تُشير إلى <body>

وضحنا ذلك في الشيفرة الآتية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

DocumentType {nodeType=10,  الناتج
// ownerDocument=document, ...}
console.log(document.doctype);

// <html lang="en"> الناتج
console.log(document.documentElement);

// <head> الناتج
console.log(document.head);

// <body> الناتج
console.log(document.body);

</script>
```

```
</body>
</html>
```

ملاحظات

- نوع المستند (doctype أو DTD) هو عقدة من النوع `10` أو `DOCUMENT_TYPE_NODE` والتي لا يجب أن تخلط بينها وبين `DOCUMENT_NODE` (أي الكائن `window.document` المبني من `HTMLDocument()`)، يُنشَأ نوع المستند من الدالة البانية `DocumentType()`.

- في متصفحات Safari و Chrome و Opera لن تظهر الخاصية `document.doctype` في قائمة `document.childNodes`.

6. الكشف عن ميزات وخصائص DOM

من الممكن عبر الدالة `document.implementation.hasFeature()` أن نعرف ما هي الميزات التي يدعمها المستند الحالي وما هو مستوى دعم المتصفح لتلك الميزة. على سبيل المثال، يمكننا أن نسأل المتصفح إن كان يحتوي على دعمٍ لأساس مواصفة DOM Level 3 عبر تمرير اسم الميزة وإصدارها إلى الدالة `hasFeature()`. سنسأل المتصفح في الشيفرة الآتية إن كان يحتوي على دعمٍ لمواصفة Core ذات الإصدار 2.0 و 3.0 (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>
console.log(document
                .implementation.hasFeature('Core','2.0'));
console.log(document
                .implementation.hasFeature('Core','3.0'));
</script>
</body>
</html>
```

يُعرِّف الجدول الآتي الميزات (تُسميها المواصفة «بالوحدات» [modules]) والإصدارات التي يمكنك تمريرها إلى الدالة `hasFeature()`:

الميزة	الإصدارات المدعومة
`Core`	1.0، و 2.0، و 3.0
`XML`	1.0، و 2.0، و 3.0
`HTML`	1.0، و 2.0
`Views`	2.0

StyleSheets	2.0
CSS	2.0
CSS2	2.0
Events	2.0، و 3.0
UIEvents	2.0، و 3.0
MouseEvents	2.0، و 3.0
MutationEvents	2.0، و 3.0
HTMLEvents	2.0
Range	2.0
Traversal	2.0
LS (التحميل والحفظ بين الملفات وشجرة DOM بشكلٍ متزامن)	3.0
LS-Asnc (التحميل والحفظ بين الملفات وشجرة DOM بشكلٍ غير متزامن)	3.0
Validation	3.0

ملاحظات

- لا تثق بدالة `hasFeature()` بمفردها، وما يزال عليك التحقق فعلًا من دعم الميزة بعد استعمالها.

- يمكن عبر استخدام الدالة `isSupported` الحصول على معلومات عن دعم إحدى الميزات وذلك لعقدةٍ معيّنةٍ فقط (مثلًا: `element.isSupported(feature, version)`).

- يمكنك أن تعرف ما الميزات التي يدعمها متصفحٌ ما عبر زيارة الصفحة الآتية التي ستجد فيها جدولًا يوضِّح ما هي الميزات التي يدعمها المتصفح الذي فتح الرابط السابق.

7. الحصول على مرجعية إلى العقدة الفعّالة حاليًا في المستند

عبر استعمال الخاصية `document.activeElement` سنتمكن من الحصول على مرجعية للعقدة الفعّالة (active ، أو العقدة المُركَّز عليها [focused]).

سأضبط في الشيفرة الآتية التركيز عند تحميل الصفحة إلى عقدة `<textarea>` ثم سنحصل على مرجعيةٍ لتلك العقدة باستخدام الخاصية `activeElement` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<textarea></textarea>
```

```
<script>

// <textarea> ضبط التركيز إلى
document.querySelector('textarea').focus();

// الحصول على مرجعية إلى العنصر المُفعّل
// أو المركّز عليه في المستند
// <textarea> الناتج
console.log(document.activeElement);

</script>
</body>
</html>
```

ملاحظة

يمكن لخاصية `activeElement` إعادة العناصر التي يمكن التركيز عليها (focus) فقط. فلو زرتَ صفحة ويب وبدأتَ بالضغط على زر tab فستجد أنَّ التركيز قد انتقل من عنصرٍ إلى آخر في تلك الصفحة. تلك هي العناصر التي يمكن إعادتها عند استخدام الخاصية `activeElement`.

8. تحديد إن كان هنالك تركيزٌ على عقدة المستند أو أي عقدة داخلها

من الممكن أن نعلم عبر استخدام الدالة `document.hasFocus()` إذا ركّز المستخدم على النافذة التي تحتوي على مستند HTML الحالي. فلو نفذتَ الشيفرة الآتية ثم بدلتَ التركيز إلى نافذةٍ أو لسانٍ أو تطبيقٍ آخر، فستُعيد الدالة `hasFocus()` القيمة `false` أما لو لم تنتقل إلى نافذةٍ أو لسانٍ آخر فالناتج `true` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

// إذا أبقيتَ التركيز على النافذة/اللسان التي يوجد فيها
// المستند فالناتج true، وعدا ذلك فالناتج false
setTimeout(function()
{console.log(document.hasFocus())},5000);
</script>
</body>
</html>
```

9. الخاصية document.defaultview هي اختصارٌ للكائن الرئيسي

يجب أن تدرك أن الخاصية `defaultView` هي اختصارٌ للكائن الرئيسي (head object) في JavaScript والذي يُشار إليه في بعض الأحيان بالكائن العام (global object). الكائن الرئيسي في متصفح الويب هو الكائن `window` والخاصية `defaultView` ستُشير إليه في بيئةٍ تعمل على متصفح ويب. سيُظهِر المثال الآتي قيمة الخاصية `defaultView` في متصفح (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

// الإشارة إلى الكائن الرئيسي في JavaScript
// يُفترَض أن يكون الكائن window في متصفحات الويب
console.log(document.defaultView);

</script>
</body>
</html>
```

إذا كنتَ تتعامل مع DOM خارج بيئة المتصفحات أو لم تكن بيئة JavaScript المستعملة هي متصفح ويب (أقصد أنَّك تستعمل Node.js) فستعطيك هذه الخاصية وصولًا إلى الكائن الرئيسي.

10. الحصول على مرجعية إلى المستند من أحد عناصره عبر الخاصية ownerDocument

تُعيد الخاصية `ownerDocument` عند استخدامها على عقدة مرجعيةً إلى الكائن `Document` الذي يحتوي تلك العقدة. سأشير في المثال الآتي إلى الكائن `Document` للعنصر `<body>` في مستند HTML، ثم سأشير إلى الكائن `Document` لعقدة العنصر `<body>` الموجودة في مستندٍ معروضٍ عبر `iframe`:

```
<!DOCTYPE html>
<html lang="en">
<body>
<iframe src="http://FileServedFromServerOnSameDomain.html">
</iframe>
<script>

// الحصول على الكائن window.document الذي يحتوي على
// <body> العنصر
console.log(document.body.ownerElement);


// iframe العنصر <body> الموجود في إطار
// iframe العنصر <body> الموجود في إطار
console.log(window.frames[0].document.body.ownerElement);
```

```
</script>
</body>
</html>
```

إذا استدعيت الخاصية ownerDocument على عقدة Document فالقيمة الناتجة هي null.

الفصل الثالث:

عقد العناصر

3

1. لمحة عن الكائنات HTML*Element

لدى جميع العناصر في مستند HTML خصوصيةٌ في أنها تملك دالةً بانيةً لها في JavaScript التي تُهيّئ عقدة العنصر في شجرة DOM. على سبيل المثال، تُنشَأ عقدة DOM للعنصر <a> من الدالة البانية HTMLAnchorElement(). سأريك في المثال الآتي أنَّ العنصر <a> قد أُنشِئ فعلًا من الدالة HTMLAnchorElement() (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<a></a>

<script>

 أخذ عقدة العنصر <a> من شجرة DOM والسؤال عن اسم الدالة
// البانية التي أنشأتها
// function HTMLAnchorElement() { [native code] } الناتج
console.log(document.querySelector('a').constructor);

</script>
</body>
</html>
```

النقطة التي أردتَ إيصالها في المثال السابق هي أنَّ كل عنصر في DOM مُنشَأٌ من دالةٍ بانيةٍ فريدةٍ خاصةٍ به. يجب أن تأخذ فكرةً لا بأس بها عن الدوال البانية المستخدمة في إنشاء عناصر HTML بعد تصفح هذه القائمة (لاحظ أنَّ القائمة غير كاملة):

- `HTMLHtmlElement`
- `HTMLHeadElement`
- `HTMLLinkElement`
- `HTMLTitleElement`
- `HTMLMetaElement`
- `HTMLBaseElement`
- `HTMLIsIndexElement`
- `HTMLStyleElement`
- `HTMLBodyElement`
- `HTMLFormElement`
- `HTMLSelectElement`
- `HTMLOptGroupElement`
- `HTMLOptionElement`
- `HTMLInputElement`
- `HTMLTextAreaElement`
- `HTMLButtonElement`
- `HTMLLabelElement`
- `HTMLFieldSetElement`
- `HTMLLegendElement`
- `HTMLUListElement`
- `HTMLOListElement`
- `HTMLDListElement`
- `HTMLDirectoryElement`
- `HTMLMenuElement`
- `HTMLLIElement`
- `HTMLDivElement`
- `HTMLParagraphElement`
- `HTMLHeadingElement`
- `HTMLQuoteElement`
- `HTMLPreElement`
- `HTMLBRElement`
- `HTMLBaseFontElement`
- `HTMLFontElement`

- `HTMLModElement`
- `HTMLAnchorElement`
- `HTMLImageElement`
- `HTMLObjectElement`
- `HTMLParamElement`
- `HTMLAppletElement`
- `HTMLMapElement`
- `HTMLAreaElement`
- `HTMLScriptElement`
- `HTMLTableElement`
- `HTMLTableCaptionElement`
- `HTMLTableColElement`
- `HTMLTableSectionElement`
- `HTMLTableRowElement`
- `HTMLTableCellElement`
- `HTMLFrameSetElement`
- `HTMLFrameElement`
- `HTMLIFrameElement`

ابقِ في ذهنك أنَّ الدوال البانية `HTML*Element` السابقة ترث خاصيات ودوال الكائنات `HTMLElement` و `Element` و `Node` و `Object`.

2. خاصيات ودوال كائنات HTML*Element (بما فيها الموروثة)

للحصول على معلومات دقيقة فيما يخص الخاصيات والدوال المتوافرة لكائنات `HTML*Element`، فالأفضل تجاهل المواصفة وسؤال المتصفح عنها. تفحّص المصفوفات المُنشَأة في الشيفرة الآتية التي تفصِّل ما هي الخاصيات والدوال المتوافرة لكائنات `HTML*Element` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

var anchor = document.querySelector('a');

// الخاصيات التابعة للعنصر
console.log(Object.keys(anchor).sort());

// الخاصيات التابعة للعنصر مع الخاصيات الموروثة
var documentPropertiesIncludeInherited = [];
for(var p in document){
    documentPropertiesIncludeInherited.push(p);
}
console.log(documentPropertiesIncludeInherited.sort());

// الخاصيات الموروثة فقط
var documentPropertiesOnlyInherited = [];
for(var p in document){
    if(!document.hasOwnProperty(p)){
```

```
            documentPropertiesOnlyInherited.push(p);
        }
    }
    console.log(documentPropertiesOnlyInherited.sort());

    </script>
    </body>
    </html>
```

الخاصيات المتوافرة كثيرة حتى لو استثنينا الخاصيات الموروثة. سأعرض في القائمة الآتية شيئًا من الخاصيات والدوال التابعة للكائن السابق والتي سنناقشها في سياق هذا الفصل:

- `createElement()`
- `tagName`
- `children`
- `getAttribute()`
- `setAttribute()`
- `hasAttribute()`
- `removeAttribute()`
- `classList()`
- `dataset`
- `attributes`

للحصول على قائمة كاملة، فراجع موسوعة حسوب التي تشرح الخاصيات والدوال العامة المتوافرة لأغلبية عناصر HTML.

3. إنشاء العناصر

تُنشَأ عقد العناصر (`Element`) عندما يُفسِّر المتصفحُ مستندَ HTML ويبني شجرة DOM الموافقة له اعتمادًا على محتويات المستند. لكن نستطيع بعد ذلك إنشاء عقد العناصر (من النوع `Element`) برمجيًا باستخدام الدالة `createElement()`.

سأُنشِئ في المثال الآتي عقدة العنصر `<textarea>` ثم سأضيف العقدة إلى شجرة DOM الحالية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>
// تُنشِئ HTMLTextAreaElement() الدالة البانية
// <textarea> العنصر
var elementNode = document.createElement('textarea');
document.body.appendChild(elementNode);

// DOM التأكد أنّ العنصر قد أصبح جزءًا من
console.log(document.querySelector('textarea'));
</script>
</body>
</html>
```

القيمة المُمرَّرة إلى الدالة `createElement()` هي سلسلةٌ نصيةٌ تُحدِّد نوع العنصر (أي `tagName`) الذي سيُنشَأ.

ملاحظة	ستُحوَّل القيمة المُمرَّرة إلى الدالة `createElement()` إلى حالة الأحرف الصغيرة قبل إنشاء العنصر.

4. الحصول على اسم وسم أحد العناصر

يمكن للخاصية `tagName` الوصول إلى اسم العنصر. تُعيد الخاصية `tagName` نفس القيمة التي ستُعيدها `nodeName`. كلا القيمتين ستكونان بأحرفٍ كبيرةٍ بغض النظر عن حالة أحرف الوسم (tag) عندما كُتِبَ في شيفرة HTML.

سنحصل في مثالنا هذا على اسم وسم العنصر `<a>` الموجود في شجرة DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<a href="#">Hi</a>
<script>

// الناتج A
console.log(document.querySelector('a').tagName);
```

```
// ستُعيد الخاصية nodeName نفس القيمة
// الناتج A
console.log(document.querySelector('a').nodeName);

</script>
</body>
</html>
```

5. الحصول على قائمة بخاصيات أحد العناصر وقيمها

عبر استخدامنا للخاصية `attributes` (التي ترثها عقد العناصر من الكائن `Node`) سنتمكن من الحصول على مجموعة من عقد `Attr` المُعرَّفة للعنصر المُحدَّد. القائمة المُعادة من نوع `NameNodeMap`. سأمرّ في المثال الآتي على خاصيات (attributes) العنصر وسأفصِّل قيم كل عقد `Attr` الموجودة في القائمة (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href='#' title="title" data-foo="dataFoo" class="yes"
style="margin:0;" foo="boo"></a>

<script>
```

```
var atts = document.querySelector('a').attributes;

for(var i=0; i< atts.length; i++){
    console.log(atts[i].nodeName +'='+ atts[i].nodeValue);
}

</script>
</body>
</html>
```

ملاحظات

- يجب أن تعدّ أنَّ المصفوفة المُعادة من الوصول إلى الخاصية `attributes` هي مصفوفةٌ حية (live)، وهذا يعني أنَّ محتواها قد يتغير في أي لحظة في حال تغيرت قيمة إحدى خاصيات العنصر.

- المصفوفة المُعادة ترث `NameNodeMap` مما يوفِّر دوالًا للتعامل مع المصفوفة مثل `getNamedItem()` و `setNamedItem()` و `removeNamedItem()`.

يجب أن تنظر إلى التعامل مع الخاصية `attributes` بهذه الدوال على أنه أمرٌ ثانوي وتستخدم بدلًا من ذلك الدوال `getAttribute()` و `setAttribute()` و `hasAttribute()` و `removeAttribute()`. ويُفضِّل المؤلف عدم التعامل مباشرةً مع عقد `Attr`، والفائدة الوحيدة من

استخدام الخاصية `attributes` هي الحصول على قائمة بالخاصيات الحية التابعة للعنصر.

- الخاصية `attributes` هي مجموعة (`collection`) شبيهة بالمصفوفات التي تملك خاصية `length` المتاحة للقراءة فقط.

- الخاصيات المنطقية (Boolean attributes، مثل `<option selected>foo</option>`) ستظهر في قائمة `attributes` لكن دون ارتباطها بقيمة إلا إن وفرتَ واحدةً (مثلًا: `<option selected="selected">foo</option>`).

6. الحصول على قيمة خاصية أحد العناصر أو ضبطها أو حذفها

أفضل طريقة للحصول على قيمة خاصية (attribute) ما أو ضبطها أو حذفها هي استخدام الـدوال `getAttribute()` و `setAttribute()` و `removeAttribute()` (على التـوالي وبالترتيب). سأوضِّح استخدام كل دالة من الدوال السابقة لإدارة خاصيات العناصر في المثال الآتي (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<a href='#' title="title" data-foo="dataFoo"
style="margin:0;" class="yes" foo="boo"
```

```
hidden="hidden">#link</a>
<script>

var atts = document.querySelector('a');

// removeAttribute() حذف الخاصيات عبر الدالة
atts.removeAttribute('href');
atts.removeAttribute('title');
atts.removeAttribute('style');
atts.removeAttribute('data-foo');
atts.removeAttribute('class');
atts.removeAttribute('foo');
atts.removeAttribute('hidden'); // خاصية منطقية

// setAttribute() ضبط الخاصيات عبر الدالة
atts.setAttribute('href','#');
atts.setAttribute('title','title');
atts.setAttribute('style','margin:0;');
atts.setAttribute('data-foo','dataFoo');
atts.setAttribute('class','yes');
atts.setAttribute('foo','boo');
// الخاصيات المنطقية تتطلب إسناد قيمة إلى الخاصية أيضًا
atts.setAttribute('hidden','hidden');
```

```
// ضبط الخاصيات عبر الدالة getAttribute()
console.log(atts.getAttribute('href'));
console.log(atts.getAttribute('title'));
console.log(atts.getAttribute('style'));
console.log(atts.getAttribute('data-foo'));
console.log(atts.getAttribute('class'));
console.log(atts.getAttribute('foo'));
console.log(atts.getAttribute('hidden'));

</script>
</body>
</html>
```

ملاحظات

- استخدم الدالة `removeAttribute()` لحذف الخاصيات بدلًا من ضبط قيمتها إلى `null` أو `''` عبر الدالة `setAttribute()`.

- تتوافر بعض الخاصيات (attributes) التابعة لعقد العناصر كخاصيات للكائن (properties) (مثلًا `document.body.id` أو `document.body.className`). لكن ينصح المؤلف بتجنب استخدام هذه الخاصيات مباشرةً واستعمال الدوال آنفة الذكر للتعامل معها.

7. التحقق من امتلاك العنصر على خاصية مُعيّنة

أفضل طريقة لتحديد إن كان أحد العناصر يملك خاصيةً ما هي استخدام الدالة `hasAttribute()`. سأتحقق في المثال الآتي أنَّ العنصر `<a>` له الخاصيات `href` و `title` و `style` و `data-foo` و `class` و `foo` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<a href='#' title="title" data-foo="dataFoo"
style="margin:0;" class="yes" goo></a>
<script>
var atts = document.querySelector('a');

console.log(
    atts.hasAttribute('href'),
    atts.hasAttribute('title'),
    atts.hasAttribute('style'),
    atts.hasAttribute('data-foo'),
    atts.hasAttribute('class'),
    // الحظ أنّ القيمة true بغض النظر إن كانت هنالك
    // قيمةٌ مرتبطةٌ بالخاصية
    atts.hasAttribute('goo')
)
```

```
</script>
</body>
</html>
```

ستُعيد الدالة السابقة القيمة `true` إن كان يملك العنصر الخاصيةَ حتى لو كانت دون قيمة. أي سنحصل على قيمة مُعادة من الدالة `hasAttribute()` إن استعملناها على الخاصيات المنطقية (Boolean attributes). سأتحقق في المثال الآتي أنَّ صندوق التأشير (checkbox) مُفعَّلٌ (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<input type="checkbox" checked></input>
<script>
var atts = document.querySelector('input');

// true الناتج
console.log(atts.hasAttribute('checked'));

</script>
</body>
</html>
```

8. الحصول على قائمة بقيم الخاصية class

سيُمكِّننا استخدام الخاصية `classList` الموجودة في عقد العناصر من الوصول إلى قائمة (من النوع `DOMTokenList`) لقيم الخاصية (attribute) `class` التي يسهل التعامل معها أكثر من السلسلة النصية المُعادة من `className` والمفصول بين قيمها بفراغ. سأقارن في المثال الآتي بين استخدام `classList` و `className` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="big brown bear"></div>
<script>

var elm = document.querySelector('div');

// {0="big", 1="brown", 2="bear", length=3, ...} النـاتج
console.log(elm.classList);
// 'big brown bear' النـاتج
console.log(elm.className);

</script>
</body>
</html>
```

ملاحظات

- الخاصية `classList` هي قائمة (list) شبيهة بالمصفوفات التي تملك خاصية `length` المتاحة للقراءة فقط.

- الخاصية `classList` متاحةٌ للقراءة فقط لكن يمكن تعديلها عبر استخدام الدوال `add()` و `remove()` و `contains()` و `toggle()`.

- لم يكن يدعم متصفح IE9 للخاصية `classList`، إلا أنَّ الدعم أصبح متوافرًا في IE10. هنالك الكثير من الحلول الالتفافية لهذه المشكلة.

9. إضافة وحذف القيمة الفرعية من الخاصية class

استخدام الدوال `classList.add()` و `classList.remove()` يُسهِّل تعديل قيمة الخاصية (attribute) `class`. سأبيّن في الشيفرة الآتية كيفية إضافة وحذف الفئات من الخاصية `class` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="dog"></div>
<script>

var elm = document.querySelector('div');
```

```
elm.classList.add('cat');
elm.classList.remove('dog');
// 'cat' الناتج
console.log(elm.className);

</script>
</body>
</html>
```

10. تفعيل أو تعطيل قيمة من قيم الخاصية class

عبر استعمال الدالة `classList.toggle()` سنتمكن من تفعيل أو تعطيل قيمة من قيم الخاصية `class`. وهذا يسمح لنا بإضافة قيمة إن كانت ناقصةً أو حذفها إن كانت موجودةً. سأريك في المثال الآتي كيف تعمل الدالة `toggle()` على القيمتين `visible` و `grow` وهذا يعني أنني سأحذف `visible` وسأضيف `grow` إلى قيمة الخاصية `class` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="visible"></div>

<script>
```

```
var elm = document.querySelector('div');

elm.classList.toggle('visible');
elm.classList.toggle('grow');
// 'grow' الـنـا تج
console.log(elm.className);

</script>
</body>
</html>
```

11. معرفة إن كانت الخاصية class تحتوي فئةً معيّنة

من الممكن عبر الدالة `classList.contains()` أن نُحدِّد إن كانت قيمة الخاصية `class` تحتوي على فئةٍ مُحدَّدة. سنختبر في المثال الآتي إن كانت الخاصية `class` التابعة للعنصر `<div>` تحتوي الفئة `brown` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="big brown bear"></div>
<script>
```

```
var elm = document.querySelector('div');

// true الناتج
console.log(elm.classList.contains('brown'));

</script>
</body>
</html>
```

12. الحصول على قيم الخاصيات *-data وضبطها

تحتوي الخاصية (property) `dataset` الموجودة في عقدة عنصر على كائنٍ يحتوي على جميع الخاصيات (attributes) التابعة للعنصر والتي تبدأ بالسابقة `data-*`. ولأنَّها مجرَّد كائنٍ عاديٍ فيمكننا تعديل الخاصية `dataset` وستنعكس تلك التغييرات مباشرةً على شجرة DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div data-foo-foo="foo" data-bar-bar="bar"></div>

<script>
```

```
var elm = document.querySelector('div');

// الحصول على القيم
console.log(elm.dataset.fooFoo); // 'foo'
console.log(elm.dataset.barBar); // 'bar'

// ضبط القيم
elm.dataset.gooGoo = 'goo';
// DOMStringMap {fooFoo="foo", barBar="bar", gooGoo="goo"}
console.log(elm.dataset);

// DOM كيف يبدو العنصر في شجرة
/* <div data-foo-foo="foo" data-bar-bar="bar" data-goo-
goo="goo"> */
console.log(elm);

</script>
</body>
</html>
```

ملاحظات

- تحتوي الخاصية `dataset` على نسخةٍ من البيانات ذات كلماتٍ تبدأ بأحرفٍ كبيرة. وهذا يعني أنَّ `data-foo-foo` ستحوّل إلى الخاصية `fooFoo` في الكائن `dataset` (ذي النوع `DOMStringMap`). أي أنَّ الشرطة `"-"` ستُحذف وسيُكتَب الحرف الذي يليها بحالة كبيرة.

- يمكن ببساطة حذف خاصية من خاصيات `data-*` من شجرة DOM باستخدام المعامل `delete` على مرجعية تلك الخاصية في الكائن `dataset` (مثلًا: `delete dataset.fooFoo`).

- لا يدعم متصفح IE9 الخاصية `dataset`؛ لكن هنالك حلٌّ التفافي. إلا أنَّك تستطيع استخدام الدوال التي تجري عمليات على الخاصيات مباشرةً (مثل `getAttribute('data-foo')` و `removeAttribute('data-foo')` و `setAttribute('data-foo')` و `hasAttribute('data-foo')`).

الفصل الرابع:

تحديد عقد العناصر

3

1. تحديد عقدة معيّنة

أشهر الدوال للحصول على مرجعية لعقدة عنصر وحيدة هي:

- `querySelector()`
- `getElementById()`

سأريك استخدمهما في الشيفرة الآتية لتحديد عقدة عنصر من مستند HTML (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li>Hello</li>
<li>big</li>
<li>bad</li>
<li id="last">world</li>
</ul>

<script>
// Hello الناتج
console.log(document.querySelector('li').textContent);
// world الناتج
console.log(document.getElementById('last').textContent);
```

```
</script>
</body>
</html>
```

الدالة `getElementById()` بسيطةٌ جدًا مقارنةً مع الدالة `querySelector()` المرنة.

تستقبل الدالة `querySelector()` معاملًا (parameter) بصيغة مُحدِّدات CSS (أي CSS selectors). ما ستفعله عادةً هو تمرير مُحدِّد CSS3 للدالة (مثلًا: `'#score>tbody>tr>td:nth-of-type(2)'`) الذي سيُستخدَم لتحديد عنصر وحيد في شجرة DOM.

ملاحظات

- ستُعيد الدالة `querySelector()` أوّل عنصر ستعثر عليه في المستند الذي يُطابِق المُحدِّد. فلو نظرتَ إلى المثال السابق فستجد أنَّنا نُحدِّد جميع عناصر `li` في المستند، لكن سيُعاد أوّل عنصر تمت مطابقته فقط.

- الدالة `querySelector()` مُعرَّفةٌ ومتاحةٌ أيضًا في عقد العناصر، مما يسمح بتضييق مجال التحديد (أي عبر استعمال سياق [context] مُخصص) إلى فرعٍ معيّن في شجرة DOM.

2. تحديد أو إنشاء قائمة من عقد العناصر

أشهر الدوال المستخدمة لإنشاء قائمة (من النوع `NodeList`) لعقد العناصر الموجودة في مستند HTML هي:

- `querySelectorAll()`
- `getElementsByTagName()`
- `getElementsByClassName()`

سنستخدمها جميعها في الشيفرة الآتية لإنشاء قائمة بعناصر `<li>` الموجودة في المستند (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li class="liClass">Hello</li>
<li class="liClass">big</li>
<li class="liClass">bad</li>
<li class="liClass">world</li>
</ul>

<script>

// ستُنشِئ جميع الدوال الآتية قائمةً بعناصر li الموجودة في DOM
console.log(document.querySelectorAll('li'));
console.log(document.getElementsByTagName('li'));
console.log(document.getElementsByClassName('liClass'));
```

```
</script>
</body>
</html>
```

إذا لم يكن ذلك واضحًا إليك، فأود أن أضيف أنَّ الدوال المستعملة في المثال السابق لا تُحدِّد عنصرًا معينًا، وإنما تُنشِئ قائمةً (من النوع `NodeList`) للعناصر التي تُطابِق ما مرَّرتَه إليها.

ملاحظات

- تعدّ القوائم المُنشَأة من الدالتين `getElementsByTagName()` و `getElementsByClassName()` على أنّها «حية» (live) وستعكس دومًا حالة المستند الفعلية حتى لو خزنّا القيمة في متغيرٍ ثم حدثنا شجرة DOM.

- لا تُعيد الدالة `querySelectorAll()` قائمةً حيةً للعناصر المُحدَّدة، وهذا يعني أنَّ القائمة المُنشَأة من الدالة `querySelectorAll()` هي «لقطة» من حالة المستند في وقت إنشاء المجموعة ولا تعكس التغييرات التي تحصل في المستند في حال تعديل شيءٍ فيه. أي أنَّ القائمة ثابتة لا تتبدل.

- الدوال `querySelectorAll()` و `getElementsByTagName()` و `getElementsByClassName()` مُعرَّفة أيضًا في عقد العناصر، مما يسمح بتضييق مجال التحديد إلى فرعٍ معيّن (أو عدّة أفرع) في شجرة DOM (مثلًا: `document.getElementById('header').getElementsByClassName('a')`).

- لم أذكر الدالة `getElementsByName()` لعدم شيوع استخدامها مقارنةً بغيرها من الدوال، لكن يجب أن تدرك وجودها وأن تعلم أنها تُحدِّد عناصر `form` و `img` و `frame` و `embed` و `object` الموجودة في المستند والتي لها نفس القيمة المُسندَة إلى الخاصية `name`.

- تمرير معامل إلى الدالة `querySelectorAll()` أو `getElementsByTagName()` يحتوي على السلسلة النصية '*' -التي تعني عادةً «كل شيء»- سيؤدي إلى إعادة قائمة بكل العناصر الموجودة في المستند.

- أبقِ في ذهنك أنَّ الخاصية `childNodes` ستُعيد قائمةً من النوع `NodeList` كما تفعل الدوال `querySelectorAll()` و `getElementsByTagName()` و `getElementsByClassName()`.

- قوائم `NodeList` هي كائناتٌ شبيهة بالمصفوفات (لكنها لا ترث الدوال الخاصة بالمصفوفات) والتي تكون الخاصية `length` فيها للقراءة فقط.

3. تحديد جميع الأبناء المباشرة لأحد العناصر

يمكننا عبر استخدام الخاصية `children` في عقدة عنصر أن نحصل على قائمة (من النوع `HTMLCollection`) لجميع عقد العناصر الأبناء المباشرة (immediate child element nodes) التي تمتلكها العقدة. سأستعمل في الشيفرة الآتية الخاصية `children` لإنشاء (أو تحديد) قائمة بجميع عناصر `<li>` الموجودة داخل العنصر `<ul>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li><strong>Hi</strong></li>
<li>there</li>
</ul>

<script>

var ulElement = document.querySelector('ul').children;

// الناتج هو قائمة (شبيهة بالمصفوفات) لجميع عقد الأبناء المباشرة
console.log(ulElement); // [<li>, <li>]

</script>
</body>
</html>
```

الحظ أنَّ الخاصية `children` ستعطينا وصلًا إلى عقد العناصر الأبناء المباشرة فقط مع استثناء أيّة عقد لا تمثل عناصر (مثل العقد النصية). إذا لم يكن يملك العنصر أبناءً فستُعيد الخاصية `children` قائمةً فارغةً.

ملاحظات

- القوائم من النوع `HTMLCollection` تحتوي على العناصر بنفس ترتيب ورودها في المستند (أو في شجرة DOM).

- قوائم `HTMLCollection` هي قوائم حية، مما يعني أنها ستعكس أيّة تغييرات تُجرى على المستند ديناميكيًا.

4. تحديد العناصر مع توفير سياق للبحث

نصل عادةً إلى الدوال `querySelector()` و `querySelectorAll()` و `getElementsByTagName()` و `getElementsByClassName()` من الكائن `document` لكنها معرّفة أيضًا في عقد العناصر، وهذا يسمح باستخدام تلك الدوال منها لتحديد فرع (أو أفرع) معيّنة من شجرة DOM التي نريد الحصول على النتائج بالبحث فيها. أو بعبارةٍ أخرى، يمكنك اختيار «سياق» (context) الذي تريد من تلك الدوال البحث فيه عن عقد العناصر وذلك باستدعائها من عقدة أحد العناصر (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>
<ul>
<li class="liClass">Hello</li>
```

```
<li class="liClass">big</li>
<li class="liClass">bad</li>
<li class="liClass">world</li>
</ul>
</div>

<ul>
<li class="liClass">Hello</li>
</ul>

<script>

// سنعدّ عنصر div الآتي هو سياق البحث
// مما يجعل دوال التحديد تبحث في العناصر الموجودة فيه فقط
var div = document.querySelector('div');

console.log(div.querySelector('ul'));
console.log(div.querySelectorAll('li'));
console.log(div.getElementsByTagName('li'));
console.log(div.getElementsByClassName('liClass'));

</script>
</body>
</html>
```

لن تعمل تلك الدوال على شجرة DOM المفسّرة من المتصفح فحسب، وإنما ستعمل أيضًا على جميع بُنى DOM التي أنشأناها برمجيًا (مثال حي):

```
<!DOCTYPE html>
<html lang="en"><body>
<script>

// إنشاء بنية DOM
var divElm = document.createElement('div');
var ulElm = document.createElement('ul');
var liElm = document.createElement('li');
liElm.setAttribute('class','liClass');
ulElm.appendChild(liElm);
divElm.appendChild(ulElm);

// استخدام دوال التحديد على بنية DOM التي أنشأناها برمجيًا
console.log(divElm.querySelector('ul'));
console.log(divElm.querySelectorAll('li'));
console.log(divElm.getElementsByTagName('li'));
console.log(divElm.getElementsByClassName('liClass'));
</script>
</body></html>
```

5. قوائم مضبوطة مسبقًا تضمّ عددًا من عقد العناصر

يجب أن تدرك وجود بعض القوائم الشبيهة بالمصفوفات والتي تكون مضبوطةً مسبقًا لك والتي تحتوي على عقد العناصر الموجودة في مستند HTML. سأذكر هنا بعضها (هذه القائمة غير كاملة) والتي يجب أن تعرف عن وجودها.

- `document.forms`: جميع عناصر النماذج `<form>` في مستند HTML.
- `document.images`: جميع الصور `<img>` في مستند HTML.
- `document.links`: جميع الروابط `<a>` الموجودة في مستند HTML.
- `document.scripts`: جميع عناصر `<script>` الموجودة في مستند HTML.
- `document.styleSheets`: جميع كائنات `<link>` أو `<style>` الموجودة في مستند HTML.

ملاحظات

- تُنشَأ القوائم المضبوطة مسبقًا من الكائن `HTMLCollection`، ما عدا `document.styleSheets` إذ يستخدم الكائن `StyleSheetList`.
- القوائم من النوع `HTMLCollection` هي قوائم حية مَثَلُها كَمَثَلِ القوائم من النوع `NodeList`.

6. التحقق من أنَّ أحد العناصر سيُحدَّد عبر تعبير تحديد

نستطيع أن نَعرف عبر الدالة `matches()` إن كان سيُحدَّد أحد العناصر في حال استعملنا تعبير تحديد. فلنقل على سبيل المثال أننا نريد معرفة إن كان العنصر `<li>` هو أوّل ابن للعنصر `<ul>`، سنفعل ذلك في الشيفرة الآتية بتحديد العنصر `<li>` داخل `<ul>` ثم سؤاله إن كان سيُطابِق المُحدِّد `li:first-child`، وبسبب مطابقته له فستُعيد الدالة `matches()` القيمة `true` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<ul>
<li>Hello</li>
<li>world</li>
</ul>
<script>
// false الناتج
console.log(document.querySelector('li').matches('li:first-
child'));
</script>
</body>
</html>
```

الفصل الخامس:

الخواص البُعدية للعناصر

5

1. لمحة عن أبعاد العناصر وانزياحها وعن آلية التمرير

تُفسَّر عقد DOM وتُرسَم بأشكالٍ مرئية عند عرض مستندات HTML في متصفحات الويب. تملك العقد تمثيلًا مرئيًا تعرضه المتصفحات، ولتفحص (أو تعديل) التمثيل المرئي والأبعاد الخاصة بالعقد برمجيًا فسنستعمل واجهة برمجية مُعرَّفة في وحدة CSSOM View Module. هنالك مجموعة من الدوال الموجودة في تلك المواصفة التي توفر طرائق لتحديد أبعاد العنصر ومكانه (اعتمادًا على الإزاحة [offset]) بالإضافة إلى آليات لتعديل العقد القابلة للتمرير (scrollable). سيشرح هذا الفصل تلك الخاصيات والدوال.

ملاحظة	أغلبية تلك الخاصيات (باستثناء `scrollLeft` و `scrollTop`) تأتي من مواصفة CSSOM View Module وهي للقراءة فقط وستُحسَب في كل مرة نحاول الوصول إليها فيها، أي بعبارةٍ أخرى: تلك القيم هي قيمٌ حية (live).

2. الحصول على قيم الإزاحة نسبةً إلى offsetParent

باستخدام الخاصيتين `offserTop` و `offsetLeft` سنتمكن من معرفة مقدار الإزاحة بالبكسل نسبةً إلى عقدة العنصر الموجودة في `offsetParent`. هاتان الخاصيتان ستعطياننا المسافة مقدرةً بالبكسل من زاوية الإطار الخارجي العليا اليسرى إلى زاوية الإطار الداخلي العليا اليسرى للعنصر الموجود في `offsetParent`. تُحدَّد قيمة `offsetParent` عبر البحث عن أقرب أب (مباشر أو غير مباشر) للعنصر الحالي الذي يملك خاصية `position` (في CSS) قيمتها لا تساوي `static`. إذا لم يُعثر على هكذا عنصر فستُشير الخاصية `offsetParent` إلى العنصر

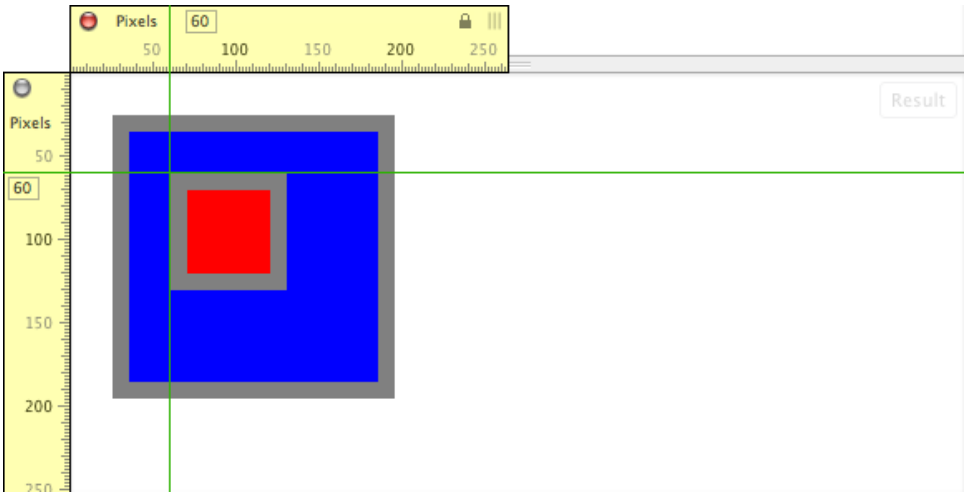
<body>. فلو كان عندنا جدول وكانت قيمة الخاصية position في CSS لعنصر <td> أو <tr> أو <table> لا تساوي static، فسيصبح ذاك العنصر هو offsetParent.

لنتحقق من أنَّ الخاصتين offsetTop و offsetLeft توفران قيمًا صحيحةً. يجب أن تخبرنا الخاصيتان offsetLeft و offsetTop في الشيفرة الآتية أنَّ العنصر ذا الخاصية id بقيمة red يبعد 60 بكسل عن الزاوية العليا اليسرى من العنصر الموجود في offsetParent وهو العنصر <body> في هذا المثال (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
body{margin:0;}
#blue{height:100px;width:100px;background-
color:blue;border:10px solid gray; padding:25px;margin:25px;}
#red{height:50px;width:50px;background-color:red;border:10px
solid gray;}
</style>
</head>
<body>

<div id="blue"><div id="red"></div></div>
```

```
<script>

var div = document.querySelector('#red');

console.log(div.offsetLeft); // 60
console.log(div.offsetTop); // 60
console.log(div.offsetParent); // <body>

</script>
</body>
</html>
```

ألقِ نظرةً على الصورة الآتية التي توضِّح كيف ستبدو الشيفرة السابقة في المتصفح لتساعدك في تخيل كيف تُحدَّد قيمة الخاصيتين `offsetLeft` و `offsetTop`. العنصر `<div>` ذو اللون الأحمر يبعد `60` بكسل تمامًا عن `offsetParent`.



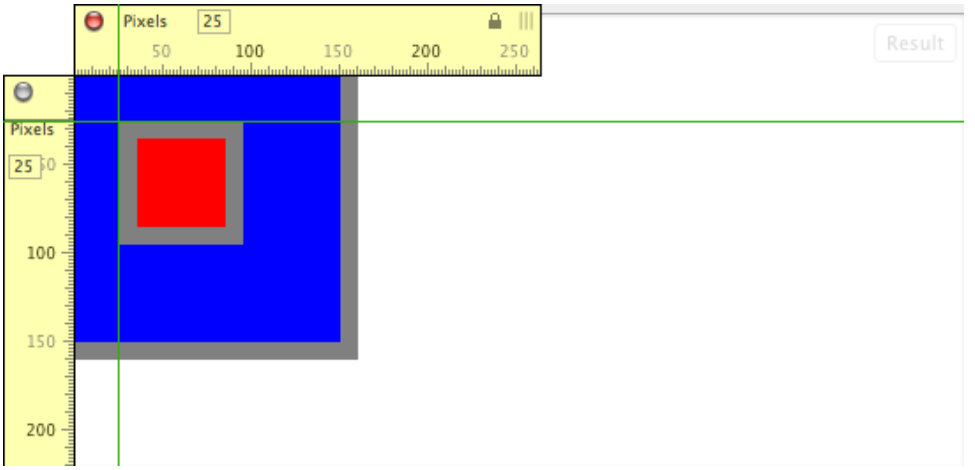
لاحظ أنني أقيس من الإطار الخارجي للعنصر `<div>` ذي اللون الأحمر إلى الإطار الداخلي لعنصر `offsetParent` (أي `<body>`).

وكما ذكرتُ سابقًا، إذا عدَّلتُ عنصر `<div>` الأزرق في المثال السابق ليصبح موضعه مطلقًا (أي ستكون لـه الخاصية `position: absolute`) فهذا سيؤدي بدوره إلى تغيير قيمة `offsetParent`. سأفعل ذلك في المثال الآتي وستتغير القيم المُعادة من الخاصيتين `offsetLeft` و `offsetTop` لتصبح 25 بكسل. وهذا لأنَّ قيمة `offsetParent` أصبحت العنصر `<div>` وليس `<body>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
```

```
#blue{height:100px;width:100px;background-
color:blue;border:10px solid gray;
padding:25px;margin:25px;position:absolute;}
#red{height:50px;width:50px;background-color:red;border:10px
solid gray;}
</style>
</head>
<body>

<div id="blue"><div id="red"></div></div>

<script>

var div = document.querySelector('#red');

console.log(div.offsetLeft); // 25
console.log(div.offsetTop); // 25
console.log(div.offsetParent); // <div id="blue">

</script>
</body>
</html>
```

الصورة الآتية المأخوذة من المتصفح ستوضِّح كيف أنَّه قد تم قياس القيمتين `offsetLeft` و `offsetTop` من العنصر `<div>` ذي اللون الأزرق.



ملاحظات

- العديد من المتصفحات تتجاوز القاعدة وتقيس بدءًا من الإطار الداخلي عندما تكون قيمة `offsetParent` هي `<body>` وكان للعنصر `<body>` (أو `<html>`) قيمة مرئية للهامش (margin) أو الحاشية (padding) أو الإطار (border).

- الخاصيات `offsetParent` و `offsetTop` و `offsetLeft` تتبع للكائن `HTMLElement`.

3. الحصول على إزاحة الإطار العلوي والسفلي والأيسر والأيمن نسبةً إلى إطار العرض

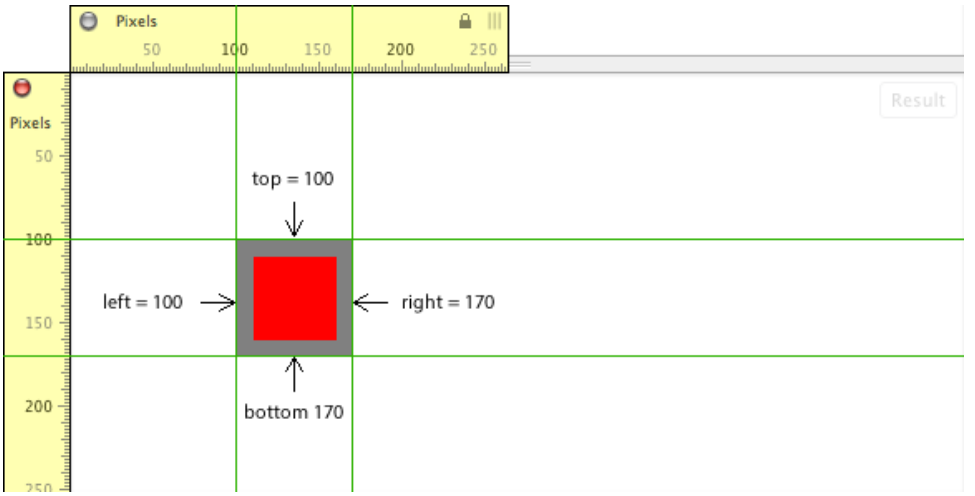
سنتمكن عبر استخدام الدالة `getBoundingClientRect()` أن نحصل على موقع الإطار الخارجي للعناصر كما هي مرسومة في إطار العرض (viewport) نسبةً إلى الزاوية العليا اليسرى من إطار العرض. وهذا يعني أنَّ الحافة اليمنى واليسرى مُقاسةٌ من الإطار الخارجي للعنصر إلى الحافة اليسرى من إطار العرض. والحافة العلوية والسفلية مُقاسةٌ من الإطار الخارجي إلى الحافة العليا لإطار العرض.

سأُنشِئ في المثال الآتي عنصر `<div>` أبعاده 50×50 بكسل وله إطار بسماكة 10 بكسل وهامش بقيمة 100 بكسل. للحصول على المسافة من كل حافة للعنصر `<div>` فسأستدعي الدالة `getBoundingClientRect()` عليه التي ستُعيد كائنًا يملك الخاصيات `top` و `right` و `bottom` و `left` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
body{margin:0;}
div{height:50px;width:50px;background-color:red;border:10px
solid gray;margin:100px;}
</style>
```

```
</head>
<body>

<div></div>

<script>

var divEdges =
document.querySelector('div').getBoundingClientRect();

console.log(divEdges.top, divEdges.right, divEdges.bottom,
divEdges.left); // '100 170 170 100'

</script>
</body>
</html>
```

تُظهِر الصورة الآتية كيفية عرض المتصفح لشيفرة السابقة مع إظهار أدوات القياس لتبيين كيف تَحسب الدالة `getBoudingClientRect()` المكان بدقة.



الحافة العليا (`top`) للإطار الخارجي للعنصر `<div>` تبعد مسافة 100 بكسل عن الحافة العليا لإطار العرض، والحافة اليمنى (`right`) للإطار الخارجي للعنصر `<div>` تبعد مسافة 170 بكسل عن الحافة اليسرى لإطار العرض. والحافة السفلى (`bottom`) للإطار الخارجي للعنصر `<div>` تبعد 170 بكسل عن الحافة العليا لإطار العرض، والحافة اليسرى (`left`) للإطار الخارجي للعنصر `<div>` تبعد 100 بكسل عن الحافة اليسرى لإطار العرض.

4. الحصول على أبعاد العنصر (الإطار والحاشية والمحتوى) في إطار العرض

صحيحٌ أنَّ الدالة `getBoundingClientRect()` تُعيد كائنًا له الخاصيات `top` و `right` و `bottom` و `left` إلا أنَّه يملك أيضًا الخاصيتين `height` و `width` واللتان تشيران إلى أبعاد العنصر، إذ تكون أبعاد العنصر هي حاصل جمع أبعاد المحتوى (content) والحاشية (padding) والإطار (border) معًا.

سأحصل على أبعاد العنصر `<div>` الموجود في شجرة DOM عبر الدالة `getBoundingClientRect()` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:25px;width:25px;background-color:red;border:25px
solid gray;padding:25px;}
</style>
</head>
<body>

<div></div>
```

```
<script>

var div =
document.querySelector('div').getBoundingClientRect();

console.log(div.height, div.width); // '125 125'
// لأن عرض الإطار 25 بكسل + 25 بكسل الحاشية + 25 للمحتوى +
// 25 للحاشية (من الطرف الآخر) + 25 إطار = 125

</script>
</body>
</html>
```

يمكن الحصول على نفس قيم الأبعاد السابقة من الخاصيتين `offsetHeight` و `offsetWidth`. سأستعمل الخاصيتين السابقتين للحصول على نفس قيم أبعاد العنصر التي توفِّرها الدالة `getBoundingClientRect()` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:25px;width:25px;background-color:red;border:25px
solid gray;padding:25px;}
```

```
</style>
</head>
<body>

<div></div>

<script>

var div = document.querySelector('div');

console.log(div.offsetHeight, div.offsetWidth); // '125 125'
// لأن عرض الإطار 25 بكسل + 25 بكسل الحاشية + 25 للمحتوى +
// 25 للحاشية (من الطرف الآخر) + 25 إطار = 125

</script>
</body>
</html>
```

5. الحصول على أبعاد العنصر (الحاشية والمحتوى) في إطار العرض دون الإطار

الخاصيتان `clientWidth` و `clientHeight` تُعيدان الأبعاد الكلّية للعنصر بجمع أبعاد محتوى العنصر مع حاشيته دون احتساب أبعاد الإطار. سأستخدم الخاصيتين السابقتين في

المثال الآتي للحصول على ارتفاع وعرض العنصر بما في ذلك حاشيته لكن باستثناء الإطار (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:25px;width:25px;background-color:red;border:25px
solid gray;padding:25px;}
</style>
</head>
<body>

<div></div>

<script>

var div = document.querySelector('div');

لأن الحاشية 25 بكسل + 25 للمحتوى + 25 للحاشية (من الطرف
// 75 = (الآخر
console.log(div.clientHeight, div.clientWidth); // '75 75'

</script>
```

```
</body>
</html>
```

6. الحصول على أعلى عنصر في إطار العرض الموجود في نقطة مُحدَّدة

يمكن عبر استعمال الدالة `elementFromPoint()` الحصول على مرجعية إلى أعلى عنصر موضعًا (أي العنصر الذي يقع أمام بقية العناصر) في مستند HTML في نقطة مُحدَّدة.

سأسأل المتصفح في المثال الآتي عن أعلى عنصر موجود على بعد 50 بكسل من أعلى إطار العرض و 50 بكسل من يسار إطار العرض. ولوجود عنصرَي `<div>` في ذلك المكان، فسيُعاد العنصر الذي يقع أمام العنصر الآخر (أي العنصر المُعرَّف لاحقًا، في حال لم تُضبَط قيمةٌ للخاصية `z-index`) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:50px;width:50px;background-
color:red;position:absolute;top:50px;left:50px;}
</style>
</head>
```

```
<body>

<div id="bottom"></div><div id="top"></div>

<script>

// <div id="top">
console.log(document.elementFromPoint(50,50));

</script>
</body>
</html>
```

7. الحصول على أبعاد العنصر الذي يتم تمريره (scroll)

الخاصيتان `scrollHeight` و `scrollWidth` تعطيانا ارتفاع وعرض العقدة التي يتم تمريرها (scrolled)، فعلى سبيل المثال، افتح أي مستند HTML الذي تتمكن من التمرير فيه داخل متصفح الويب ثم حاول الوصول إلى الخاصيتين السابقتين على عنصر `<html>` (مثلًا: `document.documentElement.scrollWidth`) أو `<body>` (مثلًا: `document.body.scrollWidth`) وستحصل على أبعاد كامل مستند HTML الذي يتم التمرير فيه.

ولمّا كنّا نستطيع جعل العنصر قابلًا للتمرير عبر CSS (أقصد عبر القاعدة `overflow: scroll`)، فلننظر إلى مثالٍ عملي. سأجعل في الشيفرة الآتية العنصر `<div>` قابلًا للتمرير، ويحتوي على عنصر `<p>` أبعاده 1000×1000 بكسل. وستُعاد أبعاد العنصر `<div>` عندما نحاول عرض قيمة الخاصيتين `scrollHeight` و `scrollWidth` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
*{margin:0;padding:0;}
div{height:100px;width:100px; overflow:auto;}
p{height:1000px;width:1000px;background-color:red;}
</style>
</head>
<body>

<div><p></p></div>

<script>

var div = document.querySelector('div');
```

```
// '1000 1000'
console.log(div.scrollHeight, div.scrollWidth);

</script>
</body>
</html>
```

ملاحظة

إذا أردت معرفة أبعاد (العرض والارتفاع) لعقدة تقع داخل منطقة قابلة للتمرير وكانت تلك العقدة أصغر من إطار العرض التابع للمنطقة القابلة للتمرير، فلا تستعمل الخاصيتين `scrollHeight` و `scrollWidth` إذ ستعطيانك أبعاد إطار العرض. فلو كانت العقدة التي يجري تمريرها أصغر من المنطقة القابلة للتمرير، فاستخدم `clientHeight` و `clientWidth` لتحديد قياس تلك العقدة.

8. الحصول على عدد البكسلات التي جرى تمريرها أو ضبط قيمتها

الخاصيتان `scrollTop` و `scrollLeft` هما خاصيتان يمكن القراءة منهما أو الكتابة إليهما إذ ستُعيدان عدد البكسلات غير الظاهرة في إطار العرض القابل للتمرير نتيجةً للتمرير في الصفحة. سأضبط في الشيفرة الآتية عنصر `<div>` قابل للتمرير الذي يحتوي على عنصر `<p>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:100px;width:100px;overflow:auto;}
p{height:1000px;width:1000px;background-color:red;}
</style>
</head>
<body>

<div><p></p></div>

<script>

var div = document.querySelector('div');

div.scrollTop = 750;
div.scrollLeft = 750;

console.log(div.scrollTop,div.scrollLeft); // '750 750'

</script>
</body>
</html>
```

لقد أجريتُ تمريرًا في عنصر برمجيًا عبر ضبط قيمة الخاصيتين scrollTop و scrollLeft إلى 750، ثم حاولت الحصول على قيمتهما، وبالطبع ستُعيدان القيمة 750 لأننا ضبطنهما إلى ذلك. القيمة 750 تُشير إلى عدد البكسلات الأعلى أو أقصى اليسار والتي لم تعد مرئيةً في إطار العرض. قد يسهل عليك فهم هاتين الخاصيتين إذا نظرتَ إليهما كقياس لعدد البكسلات غير الظاهرة في إطار العرض من الأعلى أو أقصى اليسار.

9. التمرير إلى أحد العناصر

يمكننا التمرير إلى عقدةٍ موجودةٍ في عنصرٍ قابلٍ للتمرير باستخدامنا للدالة scollIntoView(). سأُحدِّد في المثال الآتي العنصر <p> الخامس الموجود في عنصر قابل للتمرير، ثم سأستدعي الدالة scrollIntoView() عليه (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:30px;width:30px; overflow:auto;}
p{background-color:red;}
</style>
</head>
<body>

<div>
```

```
<content>
<p>1</p>
<p>2</p>
<p>3</p>
<p>4</p>
<p>5</p>
<p>6</p>
<p>7</p>
<p>8</p>
<p>9</p>
<p>10</p>
</content>
</div>

<script>

// تحديد العنصر <p>5</p> والتمرير إليه، وضعت الفهرس 4 لأنّ
// العد يبدأ من الفهرس 0 كما في المصفوفات
document.querySelector('content')
                .children[4].scrollIntoView(true);

</script>
</body>
</html>
```

عبر تمرير معامل بقيمة `true` إلى الدالة `scrollIntoView()` فسأطلب من الدالة أن تُمرِّر إلى أعلى العنصر؛ إلا أنَّ المعامل `true` ليس ضروريًا لأنَّه هو القيمة الافتراضية في الدالة، أما إذا أردتَ التمرير إلى أسفل العنصر المحدد فمرِّر القيمة `false` إلى الدالة.

الفصل السادس:

الأنماط المضمنة في عقد العناصر

6

1. لمحة عن الخاصية style

يملك كل عنصر HTML الخاصية `style` التي يمكن استخدامها لتضمين خاصيات CSS (أي أن تكون inline style) لكي تتبع إلى عنصرٍ ما. سأحاول في الشيفرة الآتية الوصول إلى الخاصية `style` في عنصر `<div>` الذي يحتوي على أكثر من قاعدة CSS في خاصية `style` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div style="background-color:red;border:1px solid
black;height:100px;width:100px;"></div>

<script>

var divStyle = document.querySelector('div').style;

// CSSStyleDeclaration {0="background-color", ...}
console.log(divStyle);

</script>
</body>
</html>
```

الحظ في الشيفرة السابقة أنَّ الخاصية `style` قد أعادت كائنًا من النوع `CSSStyleDeclaration` ولم تُعد سلسلةً نصيةً. لاحظ أيضًا عدم ذكر أيّة قواعد CSS غير مُضمَّنة (inline) في العنصر مباشرةً (أقصد قواعد CSS التي يرثها العنصر من صفحات الأنماط الأخرى) داخل الكائن `CSSStyleDeclaration`.

2. الحصول على قواعد CSS المُضمَّنة وضبطها وحذفها

يمكن تمثيل أنماط CSS كخاصية (أقصد هنا خاصية للكائن [object property]) للكائن `style` المتوافر في كائنات عقد العناصر. وهذا يوفِّر واجهةً تسمح لنا بالحصول على قيم قواعد CSS المختلفة الموجودة في عنصر HTML أو ضبطها أو حذفها.

سأضبط وأحذف وأحصل على قيمة أنماط العنصر `<div>` في المثال الآتي عبر تعديل خاصيات الكائن `style` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>

var divStyle = document.querySelector('div').style;
```

```
// ضبط الأنماط
divStyle.backgroundColor = 'red';
divStyle.border = '1px solid black';
divStyle.width = '100px';
divStyle.height = '100px';

// الحصول على الأنماط
console.log(divStyle.backgroundColor);
console.log(divStyle.border);
console.log(divStyle.width);
console.log(divStyle.height);

/* حذفها
divStyle.backgroundColor = '';
divStyle.border = '';
divStyle.width = '';
divStyle.height = '';
*/

</script>
</body>
</html>
```

ملاحظات

- أسماء الخاصيات (properties) الموجودة في الكائن `style` لا تحتوي على الشرطة – المستخدمة في أسماء خاصيات CSS، لكن التحويل بينهما بسيط، إذ عليك حذف الشرطة ووضع الحرف الذي يسبقها بالحالة الكبيرة (مثلًا: `font-size` تصبح `fontSize` و `background-image` تصبح `backgroundImage`). وفي حال كان اسم خاصيات CSS يمثِّل كلمةً محجوزةً في JavaScript فعندئذٍ أسبق الكلمة بالسابقة «css» (مثلًا: `float` تصبح `cssFloat`).

- القواعد المختصرة في CSS متوافرة أيضًا كخاصيات في JavaScript، حيث تستطيع ضبط قيمة إلى الخاصية `margin` إضافةً إلى `marginTop`.

- تذكر تضمين واحدة (unit) لكل خاصية CSS تتطلب واحدة قياس (مثلًا: `style.width = '300px';` وليس `style.width = '300';`) فعندما يُحمَّل المستند في نمط المعايير (standards mode) فسيتم تجاهل أيّة خاصية تتطلب واحدة لكنها غير موجودة، أما في نماط التجاوزات (quirks mode) فسيتم افتراض واحدة إن لم تُحدَّد.

خاصية CSS	خاصية JavaScript
`background`	`background`
`background-attachment`	`backgroundAttachment`
`background-color`	`backgroundColor`
`background-image`	`backgroundImage`

backgroundPosition	background-position
backgroundRepeat	background-repeat
border	border
borderBottom	border-bottom
borderBottomColor	border-bottom-color
borderBottomStyle	border-bottom-style
borderBottomWidth	border-bottom-width
borderColor	border-color
borderLeft	border-left
borderLeftColor	border-left-color
borderLeftStyle	border-left-style
borderLeftWidth	border-left-width
borderRight	border-right
borderRightColor	border-right-color
borderRightStyle	border-right-style
borderRightWidth	border-right-width
borderStyle	border-style
borderTop	border-top
borderTopColor	border-top-color

borderTopStyle	border-top-style
borderTopWidth	border-top-width
borderWidth	border-width
clear	clear
clip	clip
color	color
cursor	cursor
display	display
filter	filter
font	font
fontFamily	font-family
fontSize	font-size
fontVariant	font-variant
fontWeight	font-weight
height	height
left	left
letterSpacing	letter-spacing
lineHeight	line-height
listStyle	list-style

listStyleImage	list-style-image
listStylePosition	list-style-position
listStyleType	list-style-type
margin	margin
marginBottom	margin-bottom
marginLeft	margin-left
marginRight	margin-right
marginTop	margin-top
overflow	overflow
padding	padding
paddingBottom	padding-bottom
paddingLeft	padding-left
paddingRight	padding-right
paddingTop	padding-top
position	position
styleFloat	float
textAlign	text-align
textDecoration	text-decoration
textDecorationBlink	text-decoration: blink

textDecorationLineThrough	text-decoration: line-through
textDecorationNone	text-decoration: none
textDecorationOverline	text-decoration: overline
textDecorationUnderline	text-decoration: underline
textIndent	text-indent
textTransform	text-transform
top	top
verticalAlign	vertical-align
visibility	visibility
width	width
zIndex	z-index

الخاصية style هي كائن من النوع CSSStyleDeclaration وهي توفر وصولًا إلى قواعد CSS بالإضافة إلى الدوال setPropertyValue(propertyName) و getPropertyValue(propertyName, value) و removeProperty() التي تُستعمَل لتعديل مختلف قواعد CSS الموجودة في عقدة العنصر. سأحاول في الشيفرة الآتية ضبط قيم لقواعد CSS على عنصر والحصول عليها وحذفها وذلك عبر استعمال الدوال السابقة (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
</style>
</head>

<body>

<div style="background-color:green;border:1px solid
purple;"></div>

<script>

var divStyle = document.querySelector('div').style;

// ضبط القواعد
divStyle.setProperty('background-color','red');
divStyle.setProperty('border','1px solid black');
divStyle.setProperty('width','100px');
divStyle.setProperty('height','100px');
```

```
// الحصول عليها
console.log(divStyle.getPropertyValue('background-color'));
console.log(divStyle.getPropertyValue('border','1px solid
black'));
console.log(divStyle.getPropertyValue('width','100px'));
console.log(divStyle.getPropertyValue('height','100px'));

/* حذفها
divStyle.removeProperty('background-color');
divStyle.removeProperty('border');
divStyle.removeProperty('width');
divStyle.removeProperty('height');
*/

</script>
</body>
</html>
```

ملاحظة

أبقِ في ذهنك أنَّ اسم الخاصية المُمرَّر إلى الدوال السابقة (أقصد `setProperty()` و `getPropertyValue()` و `removeProperty()`) هو اسمها كما يُكتَب في css بما فيه الشرطة - (مثلًا: `background-color` وليس `backgroundColor`).

3. الحصول على جميع قواعد CSS المُضمَّنة وضبطها وحذفها

من الممكن عن استعمال الخاصية `cssText` التابعة للكائن `CSSStyleDeclaration` أو عبر استعمال الدالتين `getAttribute()` و `setAttribute()` أن نحصل أو نضبط أو نحذف جميع قواعد CSS المُضمَّنة (أي inline styles) لخاصية `style` باستخدام JavaScript. سأفعل ذلك في المثال الآتي للحصول على جميع قواعد CSS المُضمَّنة وضبطها وحذفها (عوضًا عن تغيير القواعد كلًا على حدة) وذلك على عنصر `<div>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>

var div = document.querySelector('div');
var divStyle = div.style;

// ضبط قواعد CSS عبر الخاصية cssText
divStyle.cssText = 'background-color:red;border:1px solid
black;height:100px;width:100px;';
```

```
// ثم الحصول عليها بنفس الطريقة
console.log(divStyle.cssText);

// ثم حذفها
divStyle.cssText = '';

// سنحصل على نفس الناتج تمامًا باستخدام الدالتين
// getAttribute() و setAttribute()

// setAttribute() ضبط قواعد CSS عبر الدالة
div.setAttribute('style','background-color:red;border:1px
solid black;height:100px;width:100px;');

// getAttribute() ثم الحصول عليها عبر الدالة
console.log(div.getAttribute('style'));

// removeAttribute() ثم حذفها عبر الدالة
div.removeAttribute('style');

</script>
</body>
</html>
```

ملاحظة

إن لم يكن ذلك واضحًا بالنسبة إليك، فاعلم أنَّ استبدال خاصية `style` كلها ووضع أخرى جديدة بدلًا منها هو أسرع طريقة لإجراء عدِّة تعديلات على تنسيق العنصر في آنٍ واحد.

4. الحصول على كامل الأنماط المطبقة على العنصر

تحتوي الخاصية `style` على قواعد CSS المُضمَّنة في العنصر فقط، وللحصول على جميع الأنماط المطبقة على العنصر (بما فيها تلك التي يرثها من القواعد المعرفة في العنصر `<style>` أو من صفحات الأنماط الخارجية، أو من أنماط المتصفح الافتراضية) بالإضافة إلى القواعد المُضمَّنة في العنصر، فاستخدم الدالة `getComputedStyle()` التي توفر كائن `CSSStyleDeclaration` للقراءة فقط شبيهٌ بالكائن المُعاد من الخاصية `style`. سأريك في المثال الآتي مثالًا عن قراءة أنماط أحد العناصر (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{
    background-color:red;
    border:1px solid black;
    height:100px;
    width:100px;
```

```
}
</style>
</head>

<body>

<div style="background-color:green;border:1px solid
purple;"></div>

<script>

var div = document.querySelector('div');

// الناتج rgb(0, 128, 0) أو green
// وهذه خاصية مُضمّنة في العنصر
console.log(window.getComputedStyle(div).backgroundColor);

// الناتج 1px solid rgb(128, 0, 128) أو 1px solid purple
// وهذه خاصيةٌ مُضمّنةٌ في العنصر
console.log(window.getComputedStyle(div).border);

// الناتج 100px، وهذه ليست خاصيةً مضمنةً في العنصر
console.log(window.getComputedStyle(div).height);
```

```
// الناتج 100px، وهذه ليست خاصيةً مضمنةً في العنصر
console.log(window.getComputedStyle(div).width);

</script>
</body>
</html>
```

من المهم أن تلحظ أنَّ الدالة `getComputedStyle()` تحترم قواعد وراثة الخاصيات في CSS، فمثلًا كانت قيمة `backgroundColor` للعنصر `<div>` في المثال السابق هي `green` وليست `red` لأنَّ الأنماط المُضمَّنة مباشرةً في العنصر تكون لها أولوية على غيرها، لذا أُعيدتَ القيمة المُضمَّنة مباشرةً في العنصر عند محاولة الوصول إلى الخاصية `backgroundColor` لأنَّها هي القيمة التي سيُطبِّقها المتصفح على العنصر.

ملاحظات

- لا يمكن ضبط أيّة قيم في الكائن `CSSStyleDeclaration` المُعاد من الدالة `getComputedStyles()`، إذ أنَّه للقراءة فقط.

- تُعيد الدالة `getComputedStyles()` قيم الألوان بالصيغة `rgb(#,#,#)` بغض النظر عن طريقة كتابة اللون في المستند.

- لن تُحسَب الخاصيات المختصرة في الكائن `CSSStyleDeclaration` لذا عليك استخدام الخاصيات غير المختصرة للوصول إلى قيمها (أي عليك مثلًا استخدام `marginTop` بدلًا من `margin`...).

5. تطبيق وحذف خاصيات CSS على أحد العناصر باستخدام class و id

يمكن إضافة أو حذف قواعد الأنماط المُعرَّفة في عنصر <style> أو ضمن صفحة أنماط خارجية إلى عنصرٍ باستخدام الخاصيتين class و id. وهذه هي أكثر طريقة شائعة لتعديل تنسيق العناصر.

سأستخدم في المثال الآتي الدالتين setttribute() و classList.add() لتعديل قواعد الأنماط المُطبَّقة على العنصر عبر ضبط قيمة للخاصيتين class و id. يمكن أيضًا حذف تلك القواعد عبر الدالتين removeAttribute() و classList.remove() (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>

.foo{
  background-color:red;
  padding:10px;
}
#bar{
  border:10px solid #000;
```

```
  margin:10px;
}
</style>
</head>
<body>

<div></div>

<script>

var div = document.querySelector('div');

// ضبط
div.setAttribute('id','bar');
div.classList.add('foo');

/* حذف
div.removeAttribute('id');
div.classList.remove('foo');
*/

</script>
</body>
</html>
```

الفصل السابع:

العقد النصية

7

1. لمحة عن الكائن Text

تُمثَّل النصوص الموجودة في مستند HTML ككائنات من الدالة البانية `Text()`، والتي تُنتِج عقدًا نصيةً (text nodes). وعندما يُفسَّر مستند HTML فستحوِّل النصوص الموجودة في عناصر HTML إلى عقدٍ نصيةٍ (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<p>hi</p>
<script>

// "hi" تحديد العقدة النصية
var textHi = document.querySelector('p').firstChild

console.log(textHi.constructor); // Text()

// Text {textContent="hi", length=2, wholeText="hi", ...}
console.log(textHi);

</script>
</body>
</html>
```

الفكرة التي أحاول إيصالها في المثال السابق هي أنَّ الدالة البانية `Text()` تُنشِئ العقد النصية، لكن أبقِ في ذهنك أنَّ الكائن `Text` يرث من الكائنات `CharacterData` و `Node` و `Object`.

2. خاصيات الكائن Text

للحصول على معلوماتٍ دقيقة فيما يتعلق بالخاصيات والدوال المتوافرة للكائن `Text`، فمن الأفضل تجاهل المواصفة وسؤال المتصفح عنها. تفحص المصفوفات التي ستُنشَأ في الشيفرة الآتية والتي تفصِّل ما هي الخاصيات والدوال المتوافرة للعقد النصية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<p>hi</p>
<script>
var text = document.querySelector('p').firstChild;

// الخاصيات التابعة للكائن text مباشرةً
console.log(Object.keys(text).sort());

// الخاصيات التابعة للكائن text (بما فيها الموروثة)
var textPropertiesIncludeInherited = [];
```

```
for(var p in text){
    textPropertiesIncludeInherited.push(p);
}
console.log(textPropertiesIncludeInherited.sort());

// الخاصيات الموروثة فقط
var textPropertiesOnlyInherited = [];
for(var p in text){
    if(!text.hasOwnProperty(p)){
          textPropertiesOnlyInherited.push(p);
    }
}
console.log(textPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

الخاصيات المتوافرة للكائن كثيرة حتى لو استثنينا الموروثة منها. اخترتُ قائمةً من الخاصيات والدوال جديرةً بالاهتمام، والتي سأشرحها في هذا الفصل:

- `textContent`
- `splitText()`
- `appendData()`
- `deleteData()`
- `insertData()`
- `replaceData()`

- `subStringData()`
- `data`
- `normalize()`
- `document.createTextNode()` (صحيحٌ أنها ليس خاصية تابعة للعقد النصية، إلا أنني سأشرحها هنا)

3. الفراغات تُنشِئ عقدًا نصيةً من النوع Text

عندما تُنشَأ شجرة DOM من قِبل المتصفح أو عبر الطرائق البرمجية، فستُنشَأ العقد النصية محتويةً على الفراغات أو على المحارف النصية. لا تغفل أنَّ الفراغات هي محارف في نهاية المطاف. لاحظ أنَّ الفقرة الثانية (العنصر <p>) في الشيفرة الآتية التي تحتوي على فراغٍ وحيد ستملك عقدةً نصيةً تابعةً لها، وفي حين أنَّ الفقرة الأولى لا تملك عقدةً نصيةً تابعةً لها (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<p id="p1"></p>
<p id="p2"> </p>

<script>
```

```
// null
console.log(document.querySelector('#p1')
                            .firstChild);
// #text
console.log(document.querySelector('#p2')
                            .firstChild.nodeName);
</script>
</body>
</html>
```

لا تنسَ أنَّ الفراغات والمحارف النصية تُمثَّل عادةً في شجرة DOM على أنها عقدٌ نصيةٌ، وهذا يعني أيضًا أنَّ محرف العودة إلى بداية السطر (carriage return) هو عقدةٌ نصية. سأريك ذلك في المثال الآتي الذي أظهِر فيه أنَّ محرف العودة إلى بداية السطر يُمثَّل في شجرة DOM على أنَّه عقدةٌ نصيةٌ (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<p id="p1"></p> <!--
 نعم، توجد عقدة نصية تحتوي محرف العودة إلى بداية السطر قبل
 (وبعد) هذا التعليق، بغض النظر أنّ هذا التعليق هو عقدة
منفصلة -->
```

```
<p id="p2"></p>

<script>

// Text
console.log(document.querySelector('#p1').nextSibling);

</script>
</body>
</html>
```

ففي الواقع، في كل مرة تدخِل فيها فراغًا أو أي محرف في مستند HTML فمن المرجح أنَّه سيتحول إلى عقدة نصية، وإذا فكرتَ مليًا في الأمر فستجد أنَّ مستند HTML الاعتيادي يحتوي على عددٍ كبيرٍ من الفراغات العادية ومحارف السطر الجديد والتي ستُمثَّل في شجرة DOM على أنها عقدٌ نصيةٌ، ما لم تَضغط (أو تُصغِّر) مستند HTML النهائي.

4. إنشاء وإضافة عقدة نصية من النوع Text

تُنشَأ العقد النصية تلقائيًا عندما يُفسِّر المتصفح مستند HTML ويبني شجرة DOM اعتمادًا على محتويات المستند. وبعدئذٍ نستطيع أن نُنشِئ عقدًا نصيةً من النوع `Text` برمجيًا باستخدام الدالة `createTextNode()`. سأريك في المثال الآتي كيف تُنشِئ عقدةً نصيةً وتضيف تلك العقدة إلى شجرة DOM التابعة للمستند (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>

var textNode = document.createTextNode('Hi');
document.querySelector('div').appendChild(textNode);

console.log(document.querySelector('div').innerText); // Hi

</script>
</body>
</html>
```

أبقِ في ذهنك أنَّ تستطيع أن تُضيف عقدًا نصيةً إلى عناصر وبُنى DOM المُنشَأة برمجيًا. إذ سأضيف عقدةً نصيةً داخل عنصر <p> أنشأتُه برمجيًا قبل أن أضيفه إلى شجرة DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div></div>
<script>

var elementNode = document.createElement('p');
var textNode = document.createTextNode('Hi');
elementNode.appendChild(textNode);
document.querySelector('div').appendChild(elementNode);

// <div>Hi</div>
console.log(document.querySelector('div').innerHTML);

</script>
</body>
</html>
```

5. الحصول على قيمة العقد النصية عبر data. أو nodeValue

يمكن استخراج القيمة النصية المُمثَّلة عبر عقدةٍ نصيةٍ من النوع Text باستخدام الخاصية data. أو nodeValue. ستُعيد كلا الخاصيتان السلسلةَ النصيةَ الموجودة في عقدة Text. سأستخدمهما كلاهما في المثال الآتي للحصول على القيمة النصية الموجودة ضمن العنصر <p> (مثال حي):

```
<!DOCTYPE html>
<html lang="en">

<p>Hi, <strong>user</strong></p><body>

<script>

// 'Hi,'
console.log(document.querySelector('p')
                                    .firstChild.data);
// 'Hi,'
console.log(document.querySelector('p')
                                    .firstChild.nodeValue);

</script>
</body>
</html>
```

لاحظ أنَّ العنصر <p> يحتوي على عقدتين نصيتين وعلى عقدة عنصر (أقصد)، ولم نحصل إلا على القيمة الموجودة في أوّل عقدة نصية موجودة ضمن العنصر <p>.

ملاحظة

الحصول على طول السلسلة النصية الموجودة في عقدةٍ نصيةٍ سهلٌ جدًا، وذلك عبر الوصول إلى الخاصية `length` التابعة للعقدة نفسها، أو التابعة للقيمة النصية للعقدة (إما `document.querySelector('p').firstChild.length` أو `document.querySelector('p').firstChild.data.length` و `document.querySelector('p').firstChild.nodeValue.length`).

6. تعديل العقد النصية

يـوفِّر الكـائن `CharacterData` -الـذي يـورِّث خاصـياته إلى كائنـات العقـد النصـية- الـدوال الآتية التي تستعمل لتعديل العقد النصية واستخراج قيمة فرعية منها.

- `appendData()`
- `deleteData()`
- `insertData()`
- `replaceData()`
- `subStringData()`

سأريك استخدامًا لكل واحدةٍ منها في المثال الآتي (مثال حي):

```
<!DOCTYPE html>
<html lang="en">

<p>Go big Blue Blue<body>

<script>

var pElementText = document.querySelector('p').firstChild;

// إضافة علامة تعجب
pElementText.appendData('!');
console.log(pElementText.data);

// Blue حذف أوّل كلمة
pElementText.deleteData(7,5);
console.log(pElementText.data);

// إضافة كلمة Blue مرةً أخرى
pElementText.insertData(7,'Blue ');
console.log(pElementText.data);

// Blue وضع الكلمة Bunny بدلًا من أوّل كلمة
pElementText.replaceData(7,5,'Bunny ');
console.log(pElementText.data);
```

```
// Blue Bunny استخلاص السلسلة النصية الفرعية
console.log(pElementText.substringData(7,10));

</script>
</body>
</html>
```

ملاحظة

يمكن استخدام نفس دوال المعالجة السابقة على عقد التعليقات (من النوع Comment).

7. متى تظهر عقدتان نصيتان بجوار بعضهما

عادةً، لا تتواجد عقد نصية من النوع Text بجوار بعضها لأنَّ المتصفح يُنشِئ شجرة DOM بذكاء ويدمج بين العقد النصية المتتابعة؛ لكن هنالك حالتان يكون فيهما من الممكن أن تظهر عقدتان نصيتان متتاليتان. أوّل حالةٍ واضحة، وقد رأيتها منذ قليل، وتكون عندما تحتوي العقدة النصية على عقدة عنصر (Element) (مثلًا: `<p>Hi, <strong>user</strong> welcome!</p>`) وبالتالي سيُقسَّم النص إلى مجموعتين. قد تظن مخطئًا أنَّ الموضوع معقد، إلا أنه ليس كذلك، وانظر إلى العنصر <p> في المثال الآتي ولاحظ وجود ثلاث عقد بداخله، وهي -بالترتيب- عقدةٌ نصيةٌ (Text) ثم عقد عنصر (Element) ثم عقدةٌ نصيةٌ (Text) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Hi, <strong>User</strong> welcome!</p>

<script>

var pElement = document.querySelector('p');

// 3 الناتج
console.log(pElement.childNodes.length);

// 'Hi, ' الناتج
console.log(pElement.firstChild.data);
// <strong> الناتج
console.log(pElement.firstChild.nextSibling);
// ' welcome!' الناتج
console.log(pElement.lastChild.data);

</script>
</body>
</html>
```

تحدث الحالة الثانية عندما نُضيف عقدًا نصيةً يدويًا إلى عنصرٍ أنشأناه برمجيًا. سأُنشِئ في المثال الآتي عنصر <p> وسأضيف إليه عقدتين نصيتين، مما يؤدي إلى وقوع عقدتين من النوع Text بجوار بعضهما (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div></div>
<script>
var pElementNode = document.createElement('p');
var textNodeHi = document.createTextNode('Hi ');
var textNodeUser = document.createTextNode('User');

pElementNode.appendChild(textNodeHi);
pElementNode.appendChild(textNodeUser);

document.querySelector('div').appendChild(pElementNode);

console.log(document.querySelector('div p')
                            .childNodes.length); // 2
</script>
</body>
</html>
```

8. إزالة الوسوم وإعادة جميع العقد النصية الموجودة في أحد العناصر

يمكن استعمال الخاصية `textContent` للحصول على جميع العقد النصية، بالإضافة إلى القدرة على ضبط محتويات إحدى العقد إلى قيمة معيّنة. عندما تستعمل هذه الخاصية على عقدة، فستعيد سلسلة نصية تحتوي على جميع العقد الأبناء الموجودة في العنصر الذي استدعيتَ هذه الدالة عليه. تُسهِّل هذا الخاصية كثيرًا من استخلاص جميع القيم من مستند HTML.

سأستخرج في المثال الآتي جميع النص الموجود ضمن العنصر `<body>`، لاحظ أنَّ الخاصية `textContent` لا تأخذ العقد الأبناء المباشرة فقط، وإنما تجمع محتوى جميع العقد النصية بغض النظر عن تشعبها داخل أحد العناصر الموجودة في العنصر الذي استخدمنا الخاصية `textContent` عليه.

```
<!DOCTYPE html>
<html lang="en">
<body>
<h1> Dude</h2>
<p>you <strong>rock!</strong></p>
<script>

// الناتج 'Dude you rock!' مع بعض الفراغات
console.log(document.body.textContent);
```

```
</script>
</body>
</html>
```

أما عندما تُستعمل الخاصية `textContent` لضبط المحتوى النصي الموجود ضمن عقدة، فستُحذَف أولًا جميع العقد النصية الأبناء, وستُبدَّل جميعها إلى عقدةٍ نصيةٍ وحيدةٍ من النوع `Text`. سأضع في المثال الآتي عقدةً نصيةً وحيدةً بدلًا من جميع العقد الأبناء الموجودة في العنصر `<div>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div>
<h1> Dude</h2>
<p>you <strong>rock!</strong></p>
</div>
<script>
document.body.textContent = 'You don\'t rock!';
console.log(document.body.textContent); // 'You don't rock!'
</script>
</body>
</html>
```

ملاحظات	- ستُعيد الخاصية `textContent` القيمة `null` إذا استعملناها على عقدة من النوع `document` أو `doctype`. - ستُعيد الخاصية `textContent` محتويات العناصر `<script>` و `<style>`.

9. الفرق بين textContent و innerText

تدعم أغلبية متصفحات الويب الحديثة خاصية شبيهة بخاصية `textContent` اسمها `innerText`؛ لكنهما ليستا سواءً، وعليك أن تعرف ما هي الاختلافات بينهما.

- خاصية `innerText` تدرك أنماط CSS المُطبَّقة على النص، لذا إذا كان لديك نصٌ مخفيٌ فستتجاهله الخاصية `innerText`، لكن الخاصية `textContent` ستعيده.
- لمّا كانت الخاصية `innerText` تأخذ أنماط CSS بالحسبان، فستؤدي إلى إجراء إعادة تنسيق للنص (reflow)، لكن `textContent` لا تفعل ذلك.
- الخاصية `innerText` تتجاهل العقد النصية الموجودة في عناصر `<script>` و `<style>`.
- الخاصية `innerText` ستُعاير النص (normalize) المُعاد بينما لا تفعل ذلك الخاصية `textContent`. فكِّر بخاصية `textContent` على أنها تُعيد النص الموجود حرفيًا في المستند بعد إزالة الوسوم، وهذا يتضمن الفراغات والأسطر الجديد.

10. دمج عقدتين نصيتين متتاليتين لتصبحا عقدةً واحدةً

توجد العقد النصية المتتالية عادةً عندما يُضاف النص برمجيًا إلى شجرة DOM. يمكننا استخدام الدالة `normalize()` لدمج العقد النصية المتتالية التي لا تحتوي عقد عناصر من النوع `Element`؛ سيؤدي استخدام هذه الدالة إلى دمج العقد النصية المتجاورة لتصبح عقدة وحيدة.

سأنشِئ في المثال الآتي عقدتين نصتين متتاليتين، وأضيفهما إلى شجرة DOM، ثم أستعمل الدالة `normalize()` عليهما (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div></div>
<script>

var pElementNode = document.createElement('p');
var textNodeHi = document.createTextNode('Hi');
var textNodeCody = document.createTextNode('Cody');

pElementNode.appendChild(textNodeHi);
pElementNode.appendChild(textNodeCody);

document.querySelector('div').appendChild(pElementNode);
```

```
// 2
console.log(document.querySelector('p').childNodes.length);

// دمج العقد النصية المتجاورة مع بعضها
document.querySelector('div').normalize();

// 1
console.log(document.querySelector('p').childNodes.length);

</script>
</body>
</html>
```

11. تقسيم العقد النصية

عندما تستدعى الدالة `splitText()` -التي تقبل معاملًا هو مكان التقسيم [offset]- على عقدةٍ نصيةٍ من النوع `Text` فستعدِّل محتواها (وستترك النص الموجود من أولها إلى مكان التقسيم) وتُعيد عقدةً نصيةً جديدةً تحتوي على النص المحذوف من العقدة النصية الأصلية.

لدينا في المثال الآتي عقدة نصية محتواها `Hey  Yo!` والتي سنقسمها بعد `Hey` وستبقى `Hey` موجودةً في شجرة DOM بينما ستحوّل `Yo!` إلى عقدةٍ نصيةٍ جديدةٍ تُعاد من الدالة `splitText()` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Hey Yo!</p>

<script>

// إظهار العقدة النصية الجديدة المأخوذة من شجرة DOM
console.log(document.querySelector('p')
                    .firstChild.splitText(4).data); // Yo!

// إظهار ما بقي من العقدة النصية الأصلية في DOM...
console.log(document.querySelector('p')
                    .firstChild.textContent); // Hey

</script>
</body>
</html>
```

الفصل الثامن:

عقد DocumentFragment

1. لمحة عن الكائن DocumentFragment

إنشاء واستخدام عقدة من النوع `DocumentFragment` (تمثِّل قطعة مستند) سيوفر لنا شجرة DOM منفصلة عن شجرة DOM الحية في المستند. تخيل أنَّ عقد `DocumentFragment` تُمثِّل مستندًا فارغًا الذي يحتوي على شجرة DOM لكنه موجودٌ في الذاكرة فقط، ويمكن معالجة العقد الأبناء التابعة لتلك العقد في الذاكرة بسهولة ثم إسنادها إلى شجرة DOM الحية في المستند.

2. إنشاء عقدة من النوع DocumentFragment

سأُنشِئ في الشيفرة الآتية قطعة مستند (أي عقدة `DocumentFragment`) باستخدام الدالة `createDocumentFragment()` وسأُسنِد عناصر `<li>` إليها (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>
var docFrag = document.createDocumentFragment();

["blue", "green", "red", "blue", "pink"].forEach(function(e)
{
    var li = document.createElement("li");
    li.textContent = e;
```

```
    docFrag.appendChild(li);
});

console.log(docFrag.textContent); // bluegreenredbluepink

</script>
</body>
</html>
```

استخدام documentFragment لإنشاء بُنى للعقد في الذاكرة مفيدٌ جدًا عندما نريد إضافة عقد documentFragment إلى شجرة DOM الحية.

قد تتساءل لماذا يُفضَّل استخدام documentFragment بدلًا من إنشاء عنصر (عبر createElement()) فارغ والعمل داخله لإنشاء بُنى DOM. هذه هي قائمةٌ مختصرةٌ بالاختلافات:

- قد تحتوي قطعة المستند (document fragment) على أي نوع من الشيفرات (باستثناء <body> أو <html>) بينما لا يمكن فعل ذلك عبر إنشاء عنصر.
- لا تُضاف قطعة المستند نفسها إلى شجرة DOM، وإنما ستُضاف محتوياتها (أي العقد الموجودة داخلها)؛ أما إضافة عنصر إلى المستند ستؤدي إلى إضافة العنصر نفسه إلى شجرة DOM.

- عندما تُضاف قطعة المستند إلى شجرة DOM فستنتقل العناصر الموجودة في القطعة إلى مكان إسنادها، ولن تبقى موجودةً في الذاكرة في مكان إنشائك لها، لكن هذا ليس صحيحًا لعقد العناصر التي تستخدمها مؤقتًا ثم تُسندها إلى شجرة DOM الحية.

3. إضافة DocumentFragment إلى شجرة DOM الحية

عبر تمرير وسيطٍ ذي النوع `documentFragment` إلى الدالة `appendChild()` أو `insertBefore()` فستنتقل العقد الأبناء لقطعة المستند لتصبح أبناءً للعقدة الموجودة في DOM والتي استدعيت عليها إحدى الدالتين السابقتين. سأوضِّح الشرح المبهم السابق بإنشائي لقطعة مستند، وإضافة بعض عناصر `<li>` إليها، ثم إضافة تلك العناصر إلى شجرة DOM الحية باستخدام `appendChild()` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul></ul>

<script>

var ulElm = document.querySelector('ul');
var docFrag = document.createDocumentFragment();
```

```
["blue", "green", "red", "blue", "pink"].forEach(function(e)
{
    var li = document.createElement("li");
    li.textContent = e;
    docFrag.appendChild(li);
});

ulElm.appendChild(docFrag);

/* <ul><li>blue</li><li>green</li><li>red</li><li>blue</li>
<li>pink</li></ul> */
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

ملاحظة

تمرير العقد من النوع documentFragment إلى دوال إضافة العقد في شجرة DOM سيؤدي إلى إضافة جميع العقد الأبناء الموجودة في بنية قطعة المستند، دون إضافة عقدة documentFragment نفسها.

4. استخدام innerHTML على عقد documentFragment

قد يكون إنشاء بُنى DOM في الذاكرة باستخدام دوال العقد التقليدية أمرًا مملًا وطويلًا ويأخذ وقتًا كثيرًا؛ إحدى الطرائق للالتفاف على هذه المشكلة هي إنشاء عقدة documentFragment وإضافة عنصر إلى تلك القطعة لأنَّ الخاصية innerHTML لا تعمل على عقد documentFragment مباشرةً؛ ثم استخدام الخاصية innerHTML لإضافة العناصر عبر تمرير سلسلة نصية فيها عناصر HTML إليها، وبهذا ستُنشَأ بُنية DOM من سلسلةٍ نصية. سأُنشِئ في المثال الآتي بنية DOM التي أستطيع معاملتها على أنها شجرة من العقد وليست مجرد سلسلة نصية عادية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<script>

// إنشاء عقدة <div> وقطعة مستند
var divElm = document.createElement('div');
var docFrag = document.createDocumentFragment();

// إسناد عقدة <div> إلى قطعة المستند
docFrag.appendChild(divElm);
```

```
// إنشاء بُنية DOM من سلسلة نصية
docFrag.querySelector('div').innerHTML =
'<ul><li>foo</li><li>bar</li></ul>';

// ستتحول السلسلة النصية إلى بُنية DOM، والتي أستطيع
// الوصول إليها عبر دوال مثل querySelectorAll()
// لا تنسَ أنّ كامل البُنية التي أنشأناها موجودةٌ ضمن

// عنصر <div>
console.log(docFrag.querySelectorAll('li').length); // 2

</script>
</body>
</html>
```

عندما يأتي الوقت لإسناد بُنية DOM التي أنشأناها في قطعة المستند وعنصر فقد لا ترغب بإضافة عنصر إلى شجرة DOM الحية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div></div>
```

```
<script>

// إنشاء عقدة <div> وقطعة مستند
var divElm = document.createElement('div');
var docFrag = document.createDocumentFragment();

// إسناد عقدة <div> إلى قطعة المستند
docFrag.appendChild(divElm);

// إنشاء بُنية DOM من سلسلة نصية
docFrag.querySelector('div').innerHTML =
'<ul><li>foo</li><li>bar</li></ul>';

// إسناد العقد إلى شجرة DOM بدءًا من أوّل عقدة ابن
// للعنصر <div>
document.querySelector('div')
     .appendChild(docFrag.querySelector('div').firstChild);

// <ul><li>foo</li><li>bar</li></ul>
console.log(document.querySelector('div').innerHTML);

</script>
</body>
</html>
```

ملاحظة	إضافةً إلى DocumentFragment، علينا التطلّع إلى DOMParser، الذي يستطيع تفسير سلسلة نصية تحتوي شيفرة HTML وتحويلها إلى مستند DOM.

5. الإبقاء على العناصر الموجودة في قطعة المستند عند إسنادها

عندما نُسنِد عقدة documentFragment إلى شجرة DOM الحية فستنتقل العقد الموجودة في قطعة المستند إلى شجرة DOM الحية، وفي حال أردتَ الإبقاء على محتويات القطعة في الذاكرة، أي أن تبقى العقد متوافرةً بعد إسنادها، فيمكنك نسخ عقدة documentFragment كلها باستخدام cloneNode() عند إسنادها.

سأنسخ في المثال الآتي العناصر <li> الموجودة في قطعة المستند بدلًا من نقلها إلى شجرة DOM مباشرةً، مما يترك عناصر <li> داخل عقدة documentFragment بعد الإسناد (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul></ul>

<script>
```

```
// إنشاء عنصر ul وعقدة documentFragment
var ulElm = document.querySelector('ul');
var docFrag = document.createDocumentFragment();

// إضافة عناصر li إلى قطعة المستند
["blue", "green", "red", "blue", "pink"].forEach(function(e)
{
    var li = document.createElement("li");
    li.textContent = e;
    docFrag.appendChild(li);
});

// إضافة عقدة documentFragment المنسوخة إلى عنصر ul في
// شجرة DOM الحية
ulElm.appendChild(docFrag.cloneNode(true));

/* <li>blue</li><li>green</li><li>red</li><li>blue</li>
<li>pink</li> */
console.log(document.querySelector('ul').innerHTML);

// [li,li,li,li,li]
console.log(docFrag.childNodes);
</script>
</body></html>
```

الفصل التاسع:

أنماط CSS

9

1. لمحة عن أنماط CSS

تُضاف أنماط CSS (أي CSS Style Sheets) إلى مستند HTML إما باستخدام عقدة من النوع `HTMLLinkElement` (أي العنصر `<link href="stylesheet.css" rel="stylesheet" type="text/css">`) لتطبيق صفحة أنماط خارجية، أو عبر عقدة من النوع `HTMLStyleElement` (أي العنصر `<style>`) لتعريف قواعد أنماط CSS داخل الصفحة.

لدي في مستند HTML الآتي كلا العنصرين السابقين، وسأرى ما هي الدالة البانية التي بنت تلك العقدتين (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<link id="linkElement"
href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-
min.css" rel="stylesheet" type="text/css">

<style id="styleElement">
body{background-color:#fff;}
</style>
```

```
</head>
<body>

<script>

// function HTMLLinkElement() { [native code] }
console.log(document.querySelector('#linkElement')
                                        .constructor);

// function HTMLStyleElement() { [native code] }
console.log(document.querySelector('#styleElement')
                                        .constructor);

</script>
</body>
</html>
```

بعد أن تُضاف أنماط CSS إلى الصفحة، فستُمثَّل بكائنٍ من النوع CSSStyleSheet. ستُمثَّلُ كل قاعدة CSS (مثال عن قواعد CSS: body{background-color:red;}) موجودةٍ داخل صفحة الأنماط بكائن CSSStyleRule. سأتحقق مِن أيّ دالة بانية قد أنشأت صفحة الأنماط وقواعد CSS (المُحدِّدات وخاصيات CSS والقيم المرتبطة بها) في المثال الآتي (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>
<script>
// لأن function CSSStyleSheet() { [native code] } هو الناتج
// هذا الكائن هو صفحة الأنماط نفسها
console.log(document
    .querySelector('#styleElement').sheet.constructor);

// لأن function CSSStyleRule() { [native code] } هو الناتج
// هذا الكائن هو قواعد CSS الموجودة بداخل صفحة الأنماط
console.log(document.querySelector('#styleElement')
                            .sheet.cssRules[0].constructor);

</script>
</body>
</html>
```

أبـقِ في بالـك أنَّ تحديـد العنصـر الـذي يُضـمِّن قواعـد CSS في الصـفحة (أقصـد `<link>` أو `<style>`) يختلـف عن الوصـول إلى الكـائن الفعلي (`CSSStyleSheet`) الـذي يُمثِّل صـفحة الأنماط نفسها.

2. الوصول إلى جميع أماكن تعريف أنماط CSS في شجرة DOM

تعطينا الخاصية `document.styleSheets` وصولًا إلى قائمة بجميع الكائنات التي تُمثِّل أنماط CSS (أي كائنات `CSSStyleSheet`) سواءً كانت خارجيةً (ومُضمَّنةً في المستند باستخدام العنصـر `<link>`) أو مضـافةً إلى المسـتند مباشـرةً (أقصـد العنصـر `<style>`). سأسـتخدم في المثـال الآتي الخاصـية `styleSheets` للوصـول إلى جميـع صـفحات الأنمـاط الموجـودة في المستند:

```
<!DOCTYPE html>
<html lang="en">
<head>

<link
href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-
min.css" rel="stylesheet" type="text/css">

<style>
body{background-color:red;}
</style>
```

```
</head>
<body>

<script>

console.log(document.styleSheets.length); // 2
console.log(document.styleSheets[0]); // <link>
console.log(document.styleSheets[1]); // <style>

</script>
</body>
</html>
```

ملاحظات

- قائمة `styleSheets` هي قائمةٌ حيةٌ (live list) كغيرها من القوائم المرتبطة بالعقد.

- الخاصية `length` تُعيد عدد صفحات الأنماط الموجودة في القائمة بدءًا من الفهرس 0 (أي `document.styleSheets.length`).

- صفحات الأنماط الموجودة في القائمة `styleSheets` تتضمن عادةً أيّة صفحات أنماط أُنشِئَت باستعمال العنصر `<style>` أو عبر العنصر `<link>` الذي ضُبِطَت فيه العلاقة (الخاصية `rel`) إلى `"stylesheet"`.

إضافةً إلى استعمال الخاصية `styleSheets` للوصول إلى صفحات الأنماط الموجودة في المستند، يمكن أيضًا الوصول إلى صفحة الأنماط في مستند HTML بتحديد العنصر في شجرة DOM أولًا (`<style>` أو `<link>`) ثم استخدام الخاصية `.sheet` للوصول إلى كائن `CSSStyleSheet`. سأحاول في المثال الآتي الوصول إلى أنماط CSS الموجودة في مستند HTML بتحديد العنصر الذي يُستعمَل لتضمين الأنماط في المستند، ثم استخدام الخاصية `sheet` التابعة له (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<link id="linkElement"
href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-
min.css" rel="stylesheet" type="text/css">

<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>
```

```
<script>

// الحصول على كائن CSSStyleSheet للعنصر <link> كما
// لو استعملنا document.styleSheets[0]
console.log(document.querySelector('#linkElement').sheet);

// الحصول على كائن CSSStyleSheet للعنصر <style> كما
// لو استعملنا document.styleSheets[1]
console.log(document.querySelector('#styleElement').sheet);

</script>
</body>
</html>
```

3. خاصيات ودوال الكائن CSSStyleSheet

للحصول على معلومات دقيقة فيما يتعلق بالخاصيات والدوال المتوافرة لكائنات CSSStyleSheet، فمن الأفضل تجاهل المواصفة وسؤال المتصفح عنها. انظر إلى المصفوفات المُنشَأة في هذا المثال والتي تُظهِر ما هي الخاصيات والدوال المتوافرة لكائنٍ من النوع CSSStyleSheet (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>

<script>

var styleSheet =
document.querySelector('#styleElement').sheet;

// الخاصيات التابعة للكائن styleSheet مباشرةً
console.log(Object.keys(styleSheet).sort());

// الخاصيات التابعة للكائن styleSheet بما فيها الموروثة
var styleSheetPropertiesIncludeInherited = [];
for(var p in styleSheet){
    styleSheetPropertiesIncludeInherited.push(p);
}
```

```
console.log(styleSheetPropertiesIncludeInherited.sort());

// الخاصيات الموروثة فقط
var styleSheetPropertiesOnlyInherited = [];
for(var p in styleSheet){
    if(!styleSheet.hasOwnProperty(p)){
          styleSheetPropertiesOnlyInherited.push(p);
    }
}
console.log(styleSheetPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

يمكن الوصول إلى كائن CSSStyleSheet عبر قائمة styleSheets أو عبر الخاصة .she et، ويملك الخاصيات والدوال الآتية:

- disabled
- href
- media
- ownerNode
- parentStylesheet
- title
- type
- cssRules
- ownerRule
- deleteRule
- inserRule

ملاحظة	الخاصيات `href` و `media` و `ownerNode` و `parentStylesheet` و `title` و `type` هي خاصياتٌ للقراءة فقط، أي لا تستطيع أن تضبط قيمة هذه الخاصيات مباشرةً.

4. لمحة عن الكائن CSSStyleRule

يُمثِّل الكائن `CSSStyleRule` قواعد CSS الموجودة في صفحة الأنماط. ويمكننا أن نعدّه واجهةً للتعامل مع خاصيات وقيم CSS المرتبطة بمُحدِّد (selector). سأحاول في الشيفرة الآتية الوصول برمجيًا إلى تفاصيل كل قاعدة موجودة في صفحة أنماط مُضمَّنة داخل الصفحة (العنصر `<style>`) عبر الوصول إلى الكائن `CSSStyleRule` الذي يُمثِّل قواعد CSS (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body {background-color:#fff;margin:20px;} /* هذه قاعدة CSS */
p {line-height:1.4em; color:blue;} /* هذه قاعدة CSS */
</style>

</head>
<body>
```

```
<script>

var sSheet = document.querySelector('#styleElement');

// "body { background-color: red; margin: 20px; }"
console.log(sSheet.cssRules[0].cssText);
// "p { line-height: 1.4em; color: blue; }"
console.log(sSheet.cssRules[1].cssText);

</script>
</body>
</html>
```

5. خاصيات ودوال الكائن CSSStyleRule

للحصول على معلومات دقيقة فيما يتعلق بالخاصيات والدوال المتوافرة لكائنات `CSSStyleRule`، فمن الأفضل تجاهل المواصفة وسؤال المتصفح عنها. انظر إلى المصفوفات المُنشَأة في هذا المثال والتي تُظهِر ما هي الخاصيات والدوال المتوافرة لكائنٍ من النوع `CSSStyleRule` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<style id="styleElement">
body {background-color:#fff;}
</style>

</head>
<body>

<script>

var styleSheetRule =
document.querySelector('#styleElement').sheet.cssRules;

// الخاصيات التابعة للكائن styleSheetRule مباشرةً
console.log(Object.keys(styleSheetRule).sort());

// الخاصيات التابعة للكائن styleSheetRule بما فيها الموروثة
var styleSheetRulePropertiesIncludeInherited = [];
for(var p in styleSheetRule){
    styleSheetRulePropertiesIncludeInherited.push(p);
}
console.log(styleSheetRulePropertiesIncludeInherited.sort());
```

```
// الخاصيات الموروثة فقط
var styleSheetRulePropertiesOnlyInherited = [];
for(var p in styleSheetRule){
    if(!styleSheetRule.hasOwnProperty(p)){
          styleSheetRulePropertiesOnlyInherited.push(p);
    }
}
console.log(styleSheetRulePropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

الوصول برمجيًا إلى القواعد (مثل `body{background-color:red;}`) الموجودة في صفحة أنماط أصبح ممكنًا عبر الكائن `CSSRules`، والذي يحتوي على الخاصيات الآتية:

- `cssText`
- `parentRule`
- `parentStylesSheet`
- `selectorText`
- `style`
- `type`

6. الحصول على قائمة بقواعد CSS الموجودة في صفحة

ناقشنا سابقًا في هذا الفصل القائمة `styleSheets` التي توفِّر قائمةً لصفحات الأنماط الموجودة في الصفحة. يوفِّر الكائن `CSSRules` قائمةً (من النوع `CSSRulesList`) التي تحتوي على جميع قواعد CSS (أي كائنات `CSSStyleRule`) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;margin:20px;}
p{line-height:1.4em; color:blue;}
</style>

</head>
<body>

<script>
var sSheet = document.querySelector('#styleElement').sheet;

// قائمة شبيهة بالمصفوفات تحتوي على جميع كائنات CSSRules
// التي تُمثِّل كل قاعدة CSS موجودة في صفحة الأنماط
console.log(sSheet.cssRules);
```

```
console.log(sSheet.cssRules.length); // 2

// 0 القواعد مفهرسة في قائمة CSSRules بدءًا من الفهرس
console.log(sSheet.cssRules[0]); // أوّل قاعدة
console.log(sSheet.cssRules[1]); // ثاني قاعدة

</script>
</body>
</html>
```

7. إضافة وحذف قواعد CSS الموجودة في صفحة أنماط

الدالتان `insertRule()` و `deleteRule()` توفران لنا القدرة على تعديل قواعد CSS الموجودة في صفحة أنماط برمجيًا. سأستخدم في المثال الآتي الدالة `insertRule()` لإضافة القاعدة `p {color:red}` إلى عنصر `<style>` في الفهرس 1، تذكَّر أنَّ قواعد CSS الموجودة في صفحة الأنماط مرقمةٌ بدءًا من الصفر، أي بإضافتنا للقاعدة في الفهرس 1، فستنتقل القاعدة التي كانت موجودة في الفهرس 1 (أي `p {font-size:50px;}`) إلى الفهرس 2 (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<style id="styleElement">
p{line-height:1.4em; color:blue;} /* 0 الفهرس */
p{font-size:50px;} /* 1 الفهرس */
</style>

</head>
<body>
<p>Hi</p>
<script>

// 1 إضافة قاعدة CSS جديدة في الفهرس
document.querySelector('#styleElement').sheet
                            .insertRule('p{color:red}',1);

// التحقق أنها أضيفت
console.log(document.querySelector('#styleElement')
                            .sheet.cssRules[1].cssText);

// لنحذفها
document.querySelector('#styleElement').sheet.deleteRule(1);

// لنتحقق أنها حذفت
console.log(document.querySelector('#styleElement')
                            .sheet.cssRules[1].cssText);
```

```
</script>
</body>
</html>
```

حذف أو إزالة قاعدة هو أمرٌ سهلٌ جدًا، وذلك بتمرير فهرس القاعدة التي تريد حذفها في صفحة الأنماط إلى الدالة `deleteRule()`.

ملاحظة
ليس من الشائع عمليًا تعديل أنماط CSS برمجيًا، نظرًا لصعوبة إدارة «انسيابية» (cascade) الأنماط باستخدام نظام الفهارس الرقمية (أي تحديد فهرس القاعدة دون النظر إلى محتويات صفحة الأنماط نفسها). من الأسهل التعامل مع قواعد CSS في مستند HTML قبل أن يُخدَّم إلى العميل، بدلًا من محاولة تعديل القواعد برمجيًا.

8. تعديل قيمة CSSStyleRule باستخدام الخاصية style.

كما استطعنا استخدام الخاصية `style.` لتعديل أنماط CSS المرتبطة بعقد العناصر، نستطيع أيضًا استخدام الخاصية `style` لكائنات `CSSStyleRule` التي تستطيع من خلالها تعديل الأنماط مباشرةً. سأستعمل في المثال الآتي الخاصية `style.` لضبط والحصول على قيمة قواعد CSS الموجودة في عنصر `<style>` في مستند HTML (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
p{color:blue;}
strong{color:green;}
</style>

</head>
<body>

<p>Hey <strong>Dude!</strong></p>

<script>

var styleSheet =
document.querySelector('#styleElement').sheet;

// ضبط بعض قواعد CSS في صفحة الأنماط
styleSheet.cssRules[0].style.color = 'red';
styleSheet.cssRules[1].style.color = 'purple';
```

```
// الحصول على قيمها
console.log(styleSheet.cssRules[0].style.color); // 'red'
console.log(styleSheet.cssRules[1].style.color); // 'purple'

</script>
</body>
</html>
```

9. إنشاء صفحة أنماط جديدة مُضمَّنة في مستند HTML

لإنشاء صفحة أنماط جديدة ديناميكيًا في صفحة HTML بعد تحميلها، فعلينا إنشاء عقدة للعنصر `<style>` ثم إضافة قواعد CSS إلى تلك العقدة باستعمال الخاصية `innerHTML`، ثم إضافة العقدة `<style>` إلى شجرة DOM. سأُنشِئ برمجيًا في المثال الآتي صفحة أنماط وأضيف إليها القاعدة `body {color:red;}` ثم أضيفها إلى مستند HTML (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head></head>
<body>

<p>Hey <strong>Dude!</strong></p>

<script>
```

```
var styleElm = document.createElement('style');
styleElm.innerHTML = 'body{color:red}';

// لاحظ أنّ لون النص في المستند قد تحوّل إلى الأحمر
// بعد إضافتنا لصفحة الأنماط إلى شجرة DOM
document.querySelector('head').appendChild(styleElm);

</script>
</body>
</html>
```

10. إضافة صفحة أنماط خارجية جديدة إلى مستند HTML

لإضافة ملف CSS إلى مستند HTML برمجيًا، فأنشِئ عقدةً للعنصر <link> ذاتُ الخاصيات (attributes) الملائمة، ثم أضف عنصر <link> السابق إلى شجرة DOM. سأضيف صفحة أنماط خارجية برمجيًا في المثال الآتي عبر إنشاء العنصر <link> وإضافته إلى شجرة DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head></head>
<body>
```

```
<script>

// إنشاء العنصر <link> وضبط خاصياته
var linkElm = document.createElement('link');
linkElm.setAttribute('rel', 'stylesheet');
linkElm.setAttribute('type', 'text/css');
linkElm.setAttribute('id', 'linkElement');
linkElm.setAttribute('href', 'http://yui.yahooapis.com/3.3.0/
build/cssreset/reset-min.css');

// إضافته إلى شجرة DOM
document.head.appendChild(linkElm);

// التأكد من إضافته
console.log(document.querySelector('#linkElement'));

</script>
</body>
</html>
```

11. تعطيل أو تفعيل صفحات الأنماط

من الممكن عبر استعمال الخاصية `disabled` التابعة للكائن `CSSStyleSheet` أن نُفعِّل أو نُعطِّل صفحة أنماط. سأحاول الوصول -في المثال الآتي- إلى قيمة الخاصية `disabled` التابعة لصفحتَي أنماط موجودتين في مستند HTML، ثم أعطِّلهما كلاهما باستعمال الخاصية `disabled` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<head>

<link id="linkElement"
href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-
min.css" rel="stylesheet" type="text/css">

<style id="styleElement">
body{color:red;}
</style>

</head>
<body>

<script>
```

```
// disabled الحصول على القيمة الحالية للخاصية
// 'false'
console.log(document.querySelector('#linkElement')
                                                .disabled);
// 'false'
console.log(document.querySelector('#styleElement')
                                                .disabled);

// ضبط قيمة للخاصية disabled، مما يؤدي إلى تعطيل
// جميع صفحات الأنماط الموجودة في المستند
document.querySelector('#linkElement').disabled = true;
document.querySelector('#styleElement').disabled = true;

</script>
</body>
</html>
```

الفصل العاشر:

JavaScript و DOM

1. لمحة عن تنفيذ سكربتات JavaScript في مستندات HTML

يمكن إضافة سكربتات JavaScript إلى مستند HTML بتضمين ملفات JavaScript خارجية أو بكتابة محتوى تلك السكربتات ضمن المستند على شكل عقدة نصية. لا تخلط بين تضمين JavaScript في معالجة الأحداث المتعلقة بالعناصر ضمن تعريفها (أي `<div onclick="alert('yes!')"></div>`) وبين كتابة السكربت ضمن المستند في عنصر `<script>` (أي `<script>alert('hi')</script>`).

تتطلب طريقتا تضمين شيفرات JavaScript ضمن مستند HTML استعمال عقدة العنصر `<script>`. قد يحتوي العنصر `<script>` على شيفرة JavaScript أو يمكن استخدامه لتضمين ملفات خارجية عبر الخاصية `src`. سأوضِّح طريقة استعمال كلا الطريقتين في المثال الآتي (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- تضمين لملف خارجي من نطاقٍ آخر -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9
.0/underscore-min.js"></script>
```

```
<!-- كتابة الشيفرات داخل المستند مباشرةً -->
<script>
console.log('hi');
</script>

</body>
</html>
```

ملاحظات

- من الممكن إضافة وتنفيذ شيفرة JavaScript في شجرة DOM بوضعها في خاصيةٍ تابعةٍ لأحد العناصر غرضها هو معالجة أحد الأحداث (مثلًا: `<div onclick="alert('yo')"></div>`) أو عبر استعمال بروتوكول `javascript:` (مثلًا: `<a href="javascript:alert('yo')"></a>`) لكن هذه الطرائق لم تعد شائعةً ومستخدمة في البرمجيات الحديثة.

- محاولة تضمين ملف JavaScript خارجي عبر عنصر `<script>` إضافةً إلى كتابة شيفرات محليّة فيه ستؤدي إلى تجاهل الشيفرات المحلية وتنزيل ملف JavaScript الخارجي وتنفيذه.

- يجب تفادي استعمال وسم ذاتي الإغلاق (self-closing tag) للعنصر `<script>` (أي `<script src="" />`) ما لم تكن متزمِّنًا وتتبع قواعد XHTML حرفيًا.

- لا توجد خاصيات مطلوبة للعنصر `<script>` لكن هنالك بعض الخاصيات الاختيارية مثل: `async` و `charset` و `defer` و `src` و `type`.

- ستؤدي كتابة شيفرات JavaScript في مستند HTML مباشرةً إلى إنشاء عقدة نصية، مما يسمح باستخدام الخاصيتين `innerHTML` و `textContent` للحصول على محتويات الوسم `<script>`، لكن إضافة عقدة نصية جديدة تحتوي شيفرات JavaScript إلى شجرة DOM بعد إنهاء المتصفح لتفسيره للمستند لن تؤدي إلى تنفيذ شيفرة JavaScript الجديدة، وإنما ستؤدي إلى استبدال (أو إنشاء) المحتوى النصي فقط.

- إذا احتوت شيفرة JavaScript على السلسلة النصية `'</script>'` فعليك تهريب (escape) الشرطة المائلة «/» (أي استعمال `'<\/script>'`) كي لا يظن المُفسِّر أنَّ هذه السلسلة هي وسم الإغلاق للعنصر `</script>`.

2. تُفسَّر سكربتات JavaScript بشكلٍ متزامن افتراضيًا

عندما تُفسَّر شجرة DOM ويعثر المتصفح على عنصر `<script>` فسيتوقف عن تفسير المستند ويمنع أيّة عمليات عرض أو تنزيل، ثم يُنفِّذ شيفرات JavaScript الموجودة فيه. وبسبب إيقاف المتصفح لأيّة نشاطات أخرى ولعدم إكماله لتفسير ما بقي من شجرة DOM فيُعدّ هذا السلوك على أنه تفسيرٌ متزامن للسكربتات (synchronous).

إذا كانت شيفرات JavaScript موجودة في ملف خارجي وليست مُضمَّنةً في مستند HTML، فسيزداد الطين بلةً، لضرورة تنزيل ملف JavaScript أولًا لكي يُفسَّر.

سأوضِّح في الشيفرة الآتية ماذا يحدث أثناء تحميل المتصفح للصفحة وعندما يواجه عدّة سكربتات في DOM (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- إيقاف تفسير المستند، وعدم السماح بإكماله، وتنزيل
السكربت، ثم تنفيذه، ثم إكمال تفسير المستند -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9
.0/underscore-min.js"></script>

<!-- إيقاف تفسير المستند، وعدم السماح بإكماله، وتنفيذ
السكربت، ثم إكمال تفسير المستند -->
<script>console.log('hi');</script>


</body>
</html>
```

يجب أن تلحظ في المثال السابق الاختلاف بين السكربتات الموجودة في مستند HTML والسكربتات الخارجية من ناحية ضرورة تنزيل ملف خارجي.

> **ملاحظة**
>
> السلوك الافتراضي للعنصر `<script>` الذي يمنع إكمال تفسير المستند قد يؤثر على الأداء وعلى التمثيل المرئي لصفحة الويب، فلو كان عندك عدّة عناصر `<script>` في بداية مستند HTML فلن يستطيع المتصفح القيام بأيّة مهمة (مثلًا: تفسير DOM وتنزيل الموارد اللازمة كالصور والفيديو) إلى أن يُنزِّل كل سكربت ويُنفِّذ كلًّا على حدة وبالتتالي.

3. تأجيل تنزيل وتنفيذ ملفات JavaScript الخارجية

هنالك خاصية للعنصر `<script>` اسمها `defer` التي ستؤجِّل إيقاف تفسير المستند، وستؤجِّل تنزيل وتنفيذ ملف JavaScript الخارجي إلى أن ينتهي المتصفح من تفسير المستند وصولًا إلى وسم الإغلاق للعقدة `<html>`. أي أنَّ غرض هذه الخاصيةُ بسيطٌ ألا وهو تأجيل ما سيفعله المتصفح عندما يجد ملف JavaScript خارجي إلى ما بعد انتهاء تفسير بقية المستند. سأؤجل في المثال الآتي تنفيذ سكربتات JavaScript الخارجية (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<!-- تأجيل تفسير هذا السكربت إلى ما بعد الانتهاء من تفسير بقية المستند -->
<script defer src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9.0/underscore-min.js"></script>

<!-- تأجيل تفسير هذا السكربت إلى ما بعد الانتهاء من تفسير بقية المستند -->
<script defer src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>

<!-- تأجيل تفسير هذا السكربت إلى ما بعد الانتهاء من تفسير بقية المستند -->
<script defer src="http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/3.0.6/jquery.mousewheel.min.js"></script>

<script>
// نحن نعرف أنّ jQuery غير متوافرة لأنّ هذا السطر
// يقع قبل نهاية العنصر <html>
console.log(window['jQuery'] === undefined); // true
```

```
// لكن بعد تحميل كل المستند فنستطيع أن نضمن
// jQuery تحميل وتفسير مكتبة
document.body.onload = function()
{console.log(jQuery().jquery)}; // function
</script>

</body>
</html>
```

ملاحظات

- وفقًا للمواصفة، يجب تنفيذ السكربتات المؤجلة بنفس ترتيب ورودها في المستند وقبل إطلاق الحدث `DOMContentLoaded`، لكن التزام المتصفحات الحديثة بهذه القاعدة ليس تامًا وغير موحّد فيما بينها.

- الخاصية `defer` هي خاصيةٌ منطقيةٌ (Boolean attribute) التي لا تملك قيمةً مرتبطةً بها.

- تدعم بعض المتصفحات تأجيل تنفيذ السكربتات المحلية الموجودة ضمن مستند HTML، لكن ذلك ليس شائعًا في المتصفحات الحديثة.

- سيُفتَرَض عدم استخدام الدالة `document.write()` عند تأجيل تنفيذ السكربتات الخارجية، وإلا فسيظهر تحذير ولن تعمل تلك الدالة.

4. تنزيل وتفسير سكربتات JavaScript الخارجية بشكلٍ غير متزامن

يملك العنصر <script> خاصيةً باسم async التي ستتجاوز سلوك إيقاف التفسير عند بناء شجرة DOM للصفحة والعثور على عنصر <script>. سنخبر المتصفح -عبر استعمالنا لهذه الخاصية- ألّا يوقف عملية بناء الصفحة (أي عملية تفسير DOM، بما في ذلك تنزيل الوسائط الخارجية مثل الصور وصفحات الأنماط ...إلخ.) ويتجاوز تنزيل وتنفيذ السكربتات الخارجية بالتتالي.

ما يحدث عند استخدام الخاصية async هو أنَّ الملفات ستُنزَّل بالتوازي وستُفسَّر بعد إكمال تنزيلها (بغض النظر عن ترتيب ورودها في المستند). سأعلِّق على الشيفرة الآتية موضحًا ما الذي يجري عند تفسير مستند HTML وعرضه من قِبل المتصفح (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script async
src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9
.0/underscore-min.js"></script>
```

```
<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script async
src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquer
y.min.js"></script>

<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script async
src="http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/
3.0.6/jquery.mousewheel.min.js"></script>

<script>
// ليس لدينا فكرة فيما إذا كانت
// مكتبة jQuery مُنزّلةً ومُفسّرةً أم ليس بعد…
console.log(window['jQuery'] === undefined); // true

// لكن بعد تحميل كل المستند فنستطيع أن نضمن
// تحميل وتفسير مكتبة jQuery
document.body.onload = function()
{console.log(jQuery().jquery)};
</script>
</body>
</html>
```

ملاحظات

- متصفح IE10 يدعم الخاصية async، لكن IE 9 لا يدعمها.

- الجانب السلبي الرئيسي لاستخدام الخاصية async هو أنَّ ملفات JavaScript قد لا تُفسَّر بنفس ترتيب ورودها في مستند DOM، مما قد يؤدي إلى مشاكل في إدارة الاعتماديات (أي لو كانت إحدى المكتبات تعتمد على أخرى، لكن الثانية لم تُفسَّر بعد).

- الخاصية async هي خاصيةٌ منطقيةٌ ولا تملك قيمةً مرتبطةً بها.

- سيُفتَرَض عدم استخدام الدالة document.write() عند استعمال الخاصية async، وإلا فسيظهر تحذير ولن تعمل تلك الدالة.

- لدى الخاصية async أولوية على الخاصية defer إن استعملناهما على نفس عنصر <script>.

5. ضمان تنزيل وتفسير سكربتات JavaScript الخارجية بشكلٍ غير متزامن عبر تحميل السكربتات ديناميكيًا

إحدى الطرائق الالتفافية التي تستعمل لإجبار المتصفح على تنزيل وتفسير السكربتات بشكلٍ غير متزامن (asynchronous) هي إنشاء عناصر <script> برمجيًا والتي ترتبط بملفات JavaScript الخارجية ثم إضافتها إلى شجرة DOM.

سأُنشِئ في المثال الآتي عناصر <script> برمجيًا ثم أضيفها إلى <body> مما يجبر المتصفح على تحميل السكربتات المرتبطة بعناصر <script> بشكلٍ غير متزامن (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script>
var underscoreScript = document.createElement("script");
underscoreScript.src =
"https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9.0/u
nderscore-min.js";
document.body.appendChild(underscoreScript);
</script>

<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script>
var jqueryScript = document.createElement("script");
jqueryScript.src =
"http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.mi
n.js";
document.body.appendChild(jqueryScript);
</script>
```

```
<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script>
var mouseWheelScript = document.createElement("script");
mouseWheelScript.src =
"http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/3.0.
6/jquery.mousewheel.min.js";
document.body.appendChild(mouseWheelScript);
</script>

<script>
// بعد تحميل كل المستند فنستطيع أن نضمن
// jQuery تحميل وتفسير مكتبة
document.body.onload = function()
{console.log(jQuery().jquery)};
</script>
</body>
</html>
```

ملاحظة

الجانب السلبي الرئيسي لاستخدام التحميل الديناميكي للسكربتات هو أنَّ ملفات JavaScript قد لا تُفسَّر بنفس ترتيب ورودها في مستند DOM، مما قد يؤدي إلى مشاكل في إدارة الاعتماديات.

6. معرفة متى ينتهي تحميل سكربت يُفسَّر بشكل غير متزامن

يدعم العنصر `<script>` حدثًا يدلّ أنَّ تنزيل السكربت الخارجي وتفسيره قد انتهى (وهو الحدث `onload`). سأستعمل في الشيفرة الآتية الحدث `onload` لتنبيهنا (برمجيًا) متى ينتهي تنزيل وتنفيذ ملف JavaScript (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
<script>
var underscoreScript = document.createElement("script");
underscoreScript.src =
"https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9.0/u
nderscore-min.js";
underscoreScript.onload = function(){console.log('underscsore
is loaded and exectuted');};
document.body.appendChild(underscoreScript);
</script>

<!-- لا توقف التفسير، وإنما أكمل التنزيل ثم فسِّر ونفِّذ الملف
بعد إكمال تنزيله -->
```

```
<script async
src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquer
y.min.js" onload="console.log('jQuery is loaded and
exectuted');"></script>

</body>
</html>
```

7. ضع بحسبانك مكان عناصر <script> في مستند HTML

ضع نصب عينّيك الطبيعة المتزامنة لعناصر <script>، واعلم أنَّ إضافة عناصر <script> داخل العنصر <head> في مستند HTML تؤدي إلى إشكالية في توقيت التنفيذ في حال كانت شيفرات JavaScript تعتمد على إجراء عمليات على عناصر DOM التي تلي مكان تعريفها. بعبارةٍ أخرى: إذا نفذنا سكربت JavaScript في بداية المستند وكان يحاول إجراء عمليات على شجرة DOM التي ستُعرَّف بعده، فسنحصل على خطأ في JavaScript. انظر إلى هذا المثال:

```
<!DOCTYPE html>
<html lang="en">
<head>
<!--  إيقاف التفسير ومنع المتصفح من إكمال قراءة بقية
المستند، ثم تنفيذ السكربت، ثم إكمال التفسير -->
<script>
```

```
// لا يمكننا إجراء عمليات على العنصر body الآن،
// لأنه ليس موجودًا لعدم تفسيره من المتصفح،
// أي أنه ليس موجودًا في شجرة DOM بعد
/* Uncaught TypeError: Cannot read property 'innerHTML' of
null */
console.log(document.body.innerHTML);
</script>
</head>
<body>
<strong>Hi</strong>
</body>
</html>
```

ولهذا السبب يحاول الكثير من المطورين -وأضم نفسي إليهم- وضع جميع عناصر <script> قبل وسم الإغلاق للعنصر </body>، وبهذا سنضمن أنَّ جميع عناصر شجرة DOM الموجودة قبل عنصر <script> قد فُسِّرَت وأصبح من الممكن التعامل معها عبر JavaScript، وسنتخلص من الاعتماد على الحدث ready الذي كنّا نستعمله كي لا تُفسَّر شيفراتنا قبل أن تصبح شجرة DOM جاهزةً.

8. الحصول على قائمة بعناصر <script> الموجودة في شجرة DOM

الخاصية `document.scripts` المتوافرة للكائن `document` تعطينا قائمةً (من النوع `HTMLCollection`) تضم جميع عناصر `<script>` الموجودة حاليًا في شجرة DOM. سأستعمل في المثال الآتي هذه الخاصية للوصول إلى قيمة `src` لكل سكربت موجود في المستند:

```
<!DOCTYPE html>
<html lang="en">
<body>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.9
.0/underscore-min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jque
ry.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/
1.12.1/jquery-ui.min.js"></script>
```

```
<script>
Array.prototype.slice.call(document.scripts)
                        .forEach(elm => console.log(elm.src));
</script>

</body>
</html>
```

الفصل الحادي عشر:

أحداث DOM

11

1. لمحة عن أحداث DOM

تعريف «الحدث» (event) في سياق حديثنا عن DOM هو لحظة خاصة مُعرِّفة مسبقًا في وقتٍ معيّن تحدث مرتبطةً بالكائن `document` أو `window`. هذه اللحظات تُحدَّد مسبقًا وتربط برمجيًا بوظيفةٍ معينة التي ستقع عندما يحين وقت حدوث تلك اللحظات. يمكن أن تُهيّئ تلك اللحظات اعتمادًا على حالة واجهة المستخدم (مثلًا: تم التركيز [focus] على حقلٍ في نموذج إدخال أو تم سحب [drag] أحد العناصر)، أو تغيّرت حالة البيئة التي تُشغِّل برنامج JavaScript (مثلًا: انتهى تحميل الصفحة أو انتهى تنفيذ طلبية XHR)، أو تغيرت حالة البرنامج نفسه (مثلًا: مراقبة تفاعل المستخدم مع الواجهة لمدة 30 ثانية بعد تحميل الصفحة).

يمكن ضبط الأحداث باستخدام خاصيات العناصر في HTML مباشرةً، أو عبر ضبط الخاصيات باستخدام JavaScript، أو بوساطة الدالة `addEventListener()`. سأشرح الطرائق الثلاث جميعها في المثال الآتي، إذ ستؤدي جميع الطرائق إلى إطلاق الحدث `click` في كل مرة يُضغط فيها على العنصر `<div>` باستعمال مؤشر الفأرة (مثال حي):

```
<!DOCTYPE html>
<html lang="en">

<!-- طريقة إضافة الأحداث عبر وضعها في العنصر مباشرةً -->
<body onclick="console.log('fire/trigger attribute event
handler')">
```

```
<div>click me</div>

<script>
var elementDiv = document.querySelector('div');

// طريقة ربط العنصر بحدثٍ معيّن باستعمال JavaScript
elementDiv.onclick = function(){console.log('fire/trigger
property event handler')};

// طريقة ربط العنصر بأحداثٍ معيّنة باستعمال الدالة
// addEventListener()
elementDiv.addEventListener('click',function()
{console.log('fire/trigger addEventListener')}, false);
</script>
</body>
</html>
```

لاحظ أنَّ أحد تلك الأحداث مرتبطٌ بالعنصر <body>، إن وجدّتَ أنَّ إطلاق الحدث المرتبط بالعنصر <body> بالضغط على العنصر غريبٌ فاعلم أنَّك عندما تضغط على العنصر فأنت تضغط بدورك على العنصر <body>. جرِّب أن تضغط على عنصرٍ آخر في الصفحة السابقة (أضف واحدًا للتجربة إن شئت) وسترى أنَّ الحدث click سيُطلَق على العنصر <body> بمفرده.

صحيحٌ أنَّ الطرائق الثلاث السابقة تستطيع ربط الأحداث برمجيًا إلى عناصر DOM، إلا أنَّ دالة `addEventListener()` هي الحل الأكثر تنظيمًا وكفاءة، فالخاصية المُضمَّنة داخل العناصر تخلط بين شيفرات JavaScript و HTML ومن المستحسن الفصل بينهما.

الجانب السلبي لإسناد الأحداث عبر خاصيات JavaScript هو عدم إمكانية إسناد أكثر من قيمة إلى حدثٍ معيّن. أي أنَّك لا تستطيع أن تُضيف أكثر من دالة لمعالجة أحد الأحداث المرتبطة بعنصر DOM مُعيّن. سأوضِّح ذلك في المثال الآتي عبر إسناد قيمتين إلى الخاصية `onclick`، وسنلاحظ بعد ذلك أنَّ الدالة الثانية هي التي ستُستدعى عند وقوع الحدث (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>
var elementDiv = document.querySelector('div');

// ربط العنصر بحدثٍ معيّن باستعمال الخاصية onclick
elementDiv.onclick = function(){console.log('I\'m first, but
I get overridden/replace')};
```

```
// إعادة إسناد قيمة إلى الخاصية مما يؤدي إلى
// استبدال أوّل دالة
elementDiv.onclick = function(){console.log('I win')};

</script>
</body>
</html>
```

إضافةً إلى ما سبق، تعاني دوال معالجة الأحداث المُضمَّنة في عنصر HTML مباشرةً أو المُضافة عبر خاصية JavaScript من مشاكل في المجال (scope) فقد يأتي أحدهم محاولًا استخدام سلسلة المجال (scope chain) من داخل الدالة التي ستُستدعى عند إطلاق الحدث. تُبسِّط الدالة `addEventListener()` الأمر علينا وتحل تلك الإشكاليات، ولهذا سنستعملها في بقية هذا الفصل.

ملاحظات

- تدعم عقد العناصر (`Element`) عادةً إضافة معالجات أحداث مُضمَّنة في شيفرة HTML إليها (مثلًا: `<div onclick=""></div>`) وتدعم إضافة معالجات أحداث عبر خاصيات JavaScript (مثلًا: `document.querySelector('div').onclick = function() {}`) ويُسمَح أيضًا باستعمال الدالة `addEventListener()` معها.

- تدعم العقد من النوع `Document` إضافة معالجات أحداث عبر خاصيات JavaScript (مثلًا: `document.onclick = function(){}`) وتدعم استعمال الدالة `addEventListener()` أيضًا.

- يدعم الكائن `window` إضافة معالجات أحداث مُضمَّنة في العنصر `<body>` أو `<frameset>` (مثلًا: `<body onload=""></body>`) أو عبر خاصيات JavaScript (مثلًا: `window.load = function(){}`) وباستعمال الدالة `addEventListener()`.

- كان يُشار إلى إضافة الأحداث عبر خاصيات JavaScript بالاصطلاح «أحداث في DOM من المستوى 0» (DOM level 0 events). ويُشار عادةً إلى الأحداث المُضافة بوساطة الدالة `addEventListener()` بالاصطلاح «أحداث في DOM من المستوى 2» (DOM level 2 events)، لكن ذلك غريب ومربك لعدم وجود توثيق في المواصفة للأحداث من المستوى 0 أو 1. تدعى الأحداث المُضمَّنة في عناصر HTML بالاسم «معالجات أحداث HTML» (HTML event handles).

2. أنواع أحداث DOM

سأفصِّل في الجداول الآتية أشهر الأحداث المُعرَّفة مسبقًا التي يمكن ربطها بعقد `Element` وبالكائن `document` و `window`. أبقِ في بالك أنَّه ليست جميع الأحداث قابلةً للاستعمال في العقدة أو الكائن المرتبطة به؛ ولا يعني عدم ظهور رسالة خطأ عند ربط تلك الأحداث بالعنصر أنَّ من المنطقي إضافة أحداث مثل `window.onchage`، فهي غير مصممة لتعمل مع الكائن `window` (ولا حتى إطلاق تلك الأحداث، مثل إطلاق الحدث `onchange` على الكائن `window` عبر انتقال الأحداث [bubbling]).

أ. الأحداث المرتبطة بواجهة المستخدم

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
`load`	`Event`, `UIEvent`	يُطلَق هذا الحدث عندما ينتهي تحميل المورد (صفحة HTML، أو صورة، أو ملف CSS، أو مجموعة إطارات [frameset]، أو العنصر `<object>`، أو ملف JavaScript).	`Element`, `Document`, `window`, `XMLHttpRequest`, `XMLHttpRequestUpload`	لا	لا
`unload`	`UIEvent`	يُطلَق عندما يحذف العميل المورد (المستند، أو العنصر) أو أي مورد آخر يعتمد عليه المستند (صورة، أو ملف CSS أو JS).	`window`, `<body>`, `<<frameset`	لا	لا
`abort`	`Event`, `UIEvent`	يُطلَق عندما يتوقف المورد (resource مثل الصور) عن التحميل قبل إكمال تحميله تمامًا.	`Element`, `XMLHttpRequest`, `XMLHttpRequestUpload`	نعم	لا
`error`	`Event`, `UIEvent`	يُطلَق عندما يفشل تحميل أحد الموارد أو أنَّه قد نُزِّل لكن لا يمكن تفسيره بسبب خطأ لغوي فيه، مثل الصور التالفة أو وجود خطأ بُنيوي في السكربت أو وجود خطأ في ملف XML...	`Element`, `XMLHttpRequest`, `XMLHttpRequestUpload`	نعم	لا
`resize`	`UIEvent`	يُطلَق عندما يتم إعادة تحجيم (resize) المستند. سيُطلَق هذا الحدث بعد أن تُنفَّذ	`window`, `<body>`, `<<frameset`	نعم	لا

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
		جميع التأثيرات الناجمة عن إعادة تحجيم المستند من قِبل العميل.			
`scroll`	`UIEvent`	يُطلَق عندما يُمرِّر (scroll) المستخدم داخل المستند أو داخل أحد العناصر.	`Element, Document, window`	نعم	لا
`context menu`	`MouseEvent`	يُطلَق بالضغط بالزر الأيمن للفأرة على أحد العناصر.	`Element`	نعم	نعم

ب. أحداث التركيز

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
`blur`	`FocusEvent`	يُطلَق عندما يفقد العنصر التركيز (focus) إما عبر المؤشر أو عبر الضغط على زر tab.	`Element` (ما عدا `<body>` و `<frameseet>`), `Document`	لا	لا
`focus`	`FocusEvent`	يُطلَق عندما يحصل العنصر على التركيز.	`Element` (ما عدا `<body>` و `<frameseet>`), `Document`	لا	لا
`focusin`	`FocusEvent`	يُطلَق عندما يوشك أحد العناصر أن يحصل على التركيز لكن قبل انتقال التركيز تمامًا. يقع هذا الحدث قبل	`Element`	نعم	لا

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
		الحدث `focus` مباشرةً			
`focusout`	`FocusEvent`	يُطلَق عندما يوشك أحد العناصر أن يفقد التركيز لكن قبل انتقال التركيز تمامًا. يقع هذا الحدث قبل الحدث `blur` مباشرةً	`Element`	نعم	لا

ت. أحداث النماذج

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
`change`	خاصّ بنماذج HTML	يُطلَق عندما يفقد حقل النموذج التركيز وقد تغيّرَت قيمته عن القيمة التي كانت مسندةً إليه قبل أن يحصل على التركيز.	`Element`	نعم	لا
`reset`	خاصّ بنماذج HTML	يُطلَق عندما يُعاد ضبط (reset) النموذج.	`Element`	نعم	لا
`submit`	خاصّ بنماذج HTML	يُطلَق عندما يُرسَل (submit) النموذج.	`Element`	نعم	نعم
`select`	خاصّ بنماذج HTML	يُطلَق عندما يُحدِّد المستخدم بعض النص في حقلٍ نصي، بما في ذلك `input`	`Element`	نعم	لا

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
		و `textarea`.			

ث. أحداث الفأرة

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
`click`	`MouseEvent`	يُطلَق عندما ينقر على زر الفأرة الرئيسي (أو أن يضغط المستخدم على الزر Enter) على عنصر. تُعرَّف النقرة (click) على أنها وقوع الحدثَين `mousedown` و `mouseup` في نفس المكان على الشاشة. ترتيب وقوع هذه الأحداث هو `mousedown` ثم `mouseup` ثم `click`. اعتمادًا على ضبط بيئة المتصفح، يمكن أن يقع هذا الحدث حتى لو وقع أحد الأحداث `mouseover` أو `mousemove` أو `mouseout` بين الضغط على زر المؤشر وتحرير الضغط عنه. يمكن أن يُتتبَع الحدث `click` بالحدث `dblclick`.	`Element`, `Document`, `window`	نعم	نعم
`dblclick`	`MouseEvent`	يُطلَق عندما يُنقَر على زر المؤشر مرتين	`Element`, `Document`, `window`	نعم	نعم

			متتاليتين على أحد العناصر. تعريف النقر المزدوج يختلف اعتمادًا على ضبط البيئة، لكن من الضروري أن يكون العنصر نفسه بين الأحداث `mousedown` و `mouseup` و `dblclick`. يجب أن يُطلَق هذا الحدث بعد الحدث `click` إذا صَدَفَ وأن تلاقى نقرٌ مفردٌ مع نقرٍ مزدوج في نفس الوقت، وبعد الحدث `mouseup` فيما عدا ذلك.		
`mousedown`	`MouseEvent`	يُطلَق عندما يُضغط بزر المؤشر على أحد العناصر.	`Element, Document, window`	نعم	نعم
`Mouseenter`	`MouseEvent`	يُطلَق عندما يدخل المؤشر إلى داخل حدود العنصر أو أحد العناصر الموجودة داخله. هذا الحدث شبيهٌ بالحدث `mouserover`، لكن الفرق بينهما هو أنَّ هذا الحدث لا ينتقل إلى العناصر الآباء (أي ليس bubble)، لكن يجب ألّا يقع هذا الحدث عندما يتحرك المؤشر من أحد العناصر إلى داخل حدود عنصرٍ ابن له.	`Element, Document, window`	لا	لا
`mouseleave`	`MouseEvent`	يُطلَق عندما يخرج المؤشر إلى خارج حدود العنصر أو جميع العناصر الموجودة داخله. هذا الحدث شبيهٌ بالحدث `mouseout`، لكن الفرق بينهما هو أنَّ هذا الحدث لا ينتقل إلى العناصر الآباء (أي ليس bubble)، لكن يجب ألّا يقع هذا الحدث عندما يتحرك المؤشر من	`Element, Document, window`	لا	لا

		أحد العناصر إلى داخل حدود جميع العناصر الأبناء له.			
mousemove	MouseEvent	يُطلَق عندما يتحرك المؤشر وهو ما يزال فوق العنصر. تواتر إطلاق هذا الحدث أثناء حركة الفأرة تتعلق بالجهاز والمتصفح ونظام التشغيل، لكن يجب إطلاق عدّة أحداث `mousemove` عند تحريك الفأرة تحريكًا متواصلًا، بدلًا من إطلاق حدثٍ وحيد لكل حركة مستمرة. يراعى عادةً تحديد تواتر معيّن لموازنة الأداء مع استجابة البرنامج لحركة الفأرة.	`Element, Document, window`	نعم	لا
mouseout	MouseEvent	يُطلَق هذا الحدث عندما تتحرك الفأرة خارج حدود العنصر. هذا الحدث شبيهٌ بالحدث `mouseleave` لكنه يختلف عنه في موضوع نقل الحدث إلى العناصر الآباء (bubble)، ويجب أن يُطلَق هذا الحدث أيضًا عندما يتحرك المؤشر من العنصر إلى حدود أحد العناصر الأبناء.	`Element, Document, window`	نعم	نعم
mouseup	MouseEvent	يُطلَق هذا الحدث عند تحرير الضغط على زر الفأرة فوق أحد العناصر.	`Element, Document, window`	نعم	نعم
mouseover	MouseEvent	يُطلَق عندما يُحرَّك المؤشر فوق أحد العناصر.	`Element, Document, window`	نعم	نعم

ج. الأحداث المرتبطة بدولاب الفأرة

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
`wheel` (تستخدم المتصفحات الحدث `mousewheel` لكن المواصفة تستعمل الحدث `wheel`)	`WheelEvent`	يُطلَق عندما يدور دولاب الفأرة حول أي محور، أو عندما يحاكي أي جهاز إدخال مكافئ هذا الحدث (مثل بعض أنواع أجهزة الإدخال اللوحية أو لوحة اللمس [touchpad] ...إلخ.) يمكنك العثور على معلوماتٍ مفيدة حول دعم المتصفحات هنا.	`Element`, `Document`, `Window`	نعم	نعم

ح. الأحداث المرتبطة بلوحة المفاتيح

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
`keydown`	`KeyboardEvent`	يُطلَق هذا الحدث عندما يُضغَط أحد الأزرار. يقع هذا الحدث بعد إجراء عملية ربط الزر برمزٍ معيّن (key mapping) وقبل إرسال المحرف إلى الحقل النصي. يقع هذا الحدث عند الضغط على أي زر حتى لو كان هذا الزر لا يرتبط بأحد المحارف.	`Element`, `Document`	نعم	نعم

keypress	KeyboardEvent	يُطلَق عندما يُضغط على أحد الأزرار وكان ذلك الزر مرتبطًا بأحد المحارف. يقع هذا الحدث بعد إجراء عملية ربط الزر برمزٍ معيّن لكن قبل إرسال المحرف إلى الحقل النصي.	Element, Document	نعم	نعم
keyup	KeyboardEvent	يُطلَق عندما يتم تحرير الضغط على الزر. يَتبَع هذا الحدث الحدثَين keydown و keypress.	Element, Document	نعم	نعم

خ. الأحداث المرتبطة باللمس

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
touchstart	TouchEvent	يُطلَق للإشارة إلى أنَّ المستخدم قد بدأ بلمس الشاشة اللمسية.	Element, Document, window	نعم	نعم
touchend	TouchEvent	يُطلَق للإشارة إلى أنَّ المستخدم لم يعد يلمس الشاشة.	Element, Document, window	نعم	نعم
touchmove	TouchEvent	يُطلَق عندما تُحرَّك نقطة اللمس على الشاشة.	Element, Document, window	نعم	نعم
touchenter	TouchEvent	يُطلَق للإشارة إلى أنَّ نقطة اللمس أصبحت داخل منطقة تفاعلية التي تُعرَّف من قِبل عنصر DOM.	Element, Document, window	لا	؟

toucheleave	TouchEvent	يُطلَق للإشارة إلى أنَّ نقطة اللمس أصبحت خارج المنطقة التفاعلية المُعرَّفة من عنصر DOM.	Element, Document, window	لا	؟
touchcancel	TouchEvent	يُطلَق للإشارة إلى أنَّ اللمس قد تعطل (تختلف طرق تعطيل اللمس حسب الجهاز ونظام تشغيله) مثل وقوع حدث آخر في المتصفح يؤدي إلى إلغاء اللمس، أو أنَّ نقطة اللمس قد خرجت من منطقة عرض المستند إلى منطقة أخرى قابلة للتعامل مع تفاعل المستخدم.	Element, Document, window	نعم	لا

ملاحظة

تُدعَم أحداث اللمس من متصفحات الهواتف الذكية مثل iOS و Android فقط، أو من المتصفحات التي تستطيع فيها تفعيل «وضع اللمس» (مثل Chrome).

د. الأحداث المتعلقة بالكائن window والعنصر <body> والإطارات

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
afterprint	؟	يُطلَق على الكائن مباشرةً بعد طباعة المستند المرتبط فيه، أو بعد معاينته للطباعة.	window, <body>, <frameset>	لا	لا

beforeprint	؟	يُطلَق على الكائن مباشرةً قبل طباعة المستند المرتبط فيه، أو قبل معاينته للطباعة.	window, <body>, <frameset>	لا	لا
beforeunload	؟	يُطلَق مباشرةً قبل إلغاء تحميل المستند (unload).	window, <body>, <frameset>	لا	نعم
hashchange	HashChangeEvent	يُطلَق الحدث عندما يحدث تغيير إلى جزءٍ من رابط URL (الذي يلي إشارة المربع #).	window, <body>, <frameset>	لا	لا
messsage	؟	يُطلَق عندما يُرسِل المستخدم رسالةً بين المستندات (cross-document) أو عندما تُرسَل الرسالة من الكائن Worker عبر postMessage.	window, <body>, <frameset>	لا	لا
offline	NavigatorOnLine	يُطلَق الحدث عندما يعمل المتصفح دون اتصال.	window, <body>, <frameset>	لا	لا
online	NavigatorOnLine	يُطلَق الحدث عندما يعمل المتصفح مع وجود اتصالٍ بالإنترنت.	window, <body>, <frameset>	لا	لا
pagehide	PageTransitionEvent	يُطلَق هذا الحدث عند الانتقال من الصفحة عبر تأريخ المتصفح.	window, <body>, <frameset>	لا	لا

pageshow	PageTransitionEvent	يُطلَق هذا الحدث عند الانتقال إلى الصفحة عبر تأريخ المتصفح.	window, <body>, <frameset>	لا	لا

ذ. أحداث خاصة بالكائن document

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
readystatechange	Event	يُطلَق هذا الحدث عندما تتغير قيمة readyState.	Document, XMLHttpRequest	لا	لا
DOMContentLoaded	Event	يُطلَق عندما تُفسَّر كامل الصفحة لكن قبل اكتمال تنزيل جميع الموارد.	Document	نعم	لا

ر. أحداث خاصة بالسحب والإفلات

نوع الحدث	الواجهة (interface)	الوصف	العناصر التي يمكن تطبيق الحدث عليها	الفقاعات	قابل للإلغاء
drag	DragEvent	يُطلَق على الكائن الذي يخضع لعملية سحب (drag).	Element, Document, window	نعم	نعم
dragstart	DragEvent	يُطلَق على الكائن الذي يخضع لعملية السحب عندما يبدأ المستخدم بسحب سلسلة نصية مُحدَّدة (selected) أو عنصر مُحدَّد. هذا الحدث هو أوّل حدث	Element, Document, window	نعم	نعم

		سيُطلَق عندما يبدأ المستخدم بسحب العنصر عبر الفأرة.			
dragend	DragEvent	يُطلَق على الكائن الذي يخضع لعملية السحب عند تحرير الضغط على زر الفأرة عند نهاية السحب والإفلات. الحدث dragend هو آخر حدث سيُطلَق، ويتبع الحدث dragleave الذي يُطلَق على العنصر الهدف.	Element, Document, window	نعم	لا
dragenter	DragEvent	يُطلَق على العنصر الهدف عندما يسحب المستخدم العنصر الأصلي إليه شرط إمكانية إفلاته فيه.	Element, Document, window	نعم	نعم
dragleave	DragEvent	يُطلَق على العنصر الهدف عندما يُحرِّك المستخدم العنصر الأصلي بعيدًا عنه أثناء عملية السحب.	Element, Document, window	نعم	لا
dragover	DragEvent	يُطلَق على العنصر الهدف عندما يُحرِّك المستخدم العنصر الأصلي فوق منطقة يمكن إفلاته فيها. يقع الحدث dragover على العنصر الهدف بعد الحدث dragenter.	Element, Document, window	نعم	نعم
drop	DragEvent	يُطلَق على العنصر الهدف عند تحرير زر الفأرة من الضغط أثناء عملية السحب والإفلات. الحدث drop يقع قبل الحدثين dragleave و dragend.	Element, Document, window	نعم	نعم

ملاحظات

- أنشأتُ الجداول السابق من المصادر الآتية: «Document Object Model (DOM) Level 3 Events Specification 5 User Event Module» و «DOM event reference» و «HTML Living Standard 7.1.6 Event handlers on elements, Document objects, and Window objects» و «Event compatibility tables».

- ذكرتُ في هذا القسم أشهر أنواع الأحداث، لكن اعلم أنَّ هنالك عددٌ كبيرٌ من الواجهات البرمجية (APIs) الموجودة في HTML5 التي لم أضع شيئًا عنها في هذا القسم (مثل أحداث الوسائط [media events] لعناصر `<video>` و `<audio>`، أو أحداث تغيير الحالة التابعة للكائن XMLHttpRequest).

- الأحداث `copy` و `cut` و `textinput` غير مُعرَّفة من مواصفة DOM 3 أو HTML5.

3. انتشار الأحداث

عندما يقع أحد الأحداث فسيتنشر (propagates) ذاك الحدث في شجرة DOM، مما يؤدي إلى إطلاق نفس الحدث على عقدٍ أخرى وكائنات JavaScript أخرى. قد يكون انتشار الأحدث من النمط capture (أي أنَّ انتقال الأحداث يكون من «جذع» شجرة DOM إلى «أغصانها») أو من النمط bubble (أي انتقال الأحداث من «أغصان» شجرة DOM إلى «جذعها») أو كلاهما.

سأضبط في المثال الآتي عشر دوال لمعالجة الأحداث التي ستُستدعى كلها نتيجة النقر على العنصر الموجود في مستند HTML. فعند النقر على العنصر سيبدأ النمط capture في الكائن window ثم ينتشر إلى الأسفل عبر شجرة DOM مما يؤدي إلى إطلاق الحدث click لكل كائن (أي window ثم document ثم <html> ثم <body> ثم العنصر الذي أطلَق الحدث بادئ الأمر) حتى يصل إلى العنصر الهدف. بعد أن تنتهي مرحلة capture تبدأ مرحلة «الهدف» (target phase)، عبر إطلاق الحدث في العنصر الهدف نفسه، ثم في مرحلة bubble ستنتشر الأحداث صعودًا من العنصر الهدف الذي أدى إلى إطلاق الحدث click حتى تصل إلى الكائن window (أي العنصر الهدف ثم <body> ثم <html> ثم document ثم window). بعد معرفتك للمعلومات السابقة يجب أن يكون ناتج النقر على العنصر في الشيفرة الآتية واضحًا، وهو «1,2,3,4,5,6,7,8,9,11» (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me to start event flow</div>

<script>
// لاحظ أنني أُمرِّر معاملًا منطقيًا إلى الدالة addEventListener()
// قيمته true والذي سيؤدي إلى إطلاق أحداث مرحلة capture
// كي لا تُطلَق أحداث bubble فقط
```

```
// capture مرحلة :1
window.addEventListener('click',function()
{console.log(1);},true);

// capture مرحلة :2
document.addEventListener('click',function()
{console.log(2);},true);

// capture مرحلة :3
document.documentElement.addEventListener('click',function()
{console.log(3);},true);

// capture مرحلة :4
document.body.addEventListener('click',function()
{console.log(4);},true);

// capture ضمن مرحلة تقع target مرحلة :5
document.querySelector('div').addEventListener('click',functi
on(){console.log(5);},true);

// bubble ضمن مرحلة تقع target مرحلة :6
document.querySelector('div').addEventListener('click',functi
on(){console.log(6);},false);
```

```
// bubble مرحلة :7
document.body.addEventListener('click',function()
{console.log(7);},false);

// bubble مرحلة :8
document.documentElement.addEventListener('click',function()
{console.log(8);},false);

// bubble مرحلة :9
document.addEventListener('click',function()
{console.log(9);},false);

// bubble مرحلة :10
window.addEventListener('click',function()
{console.log(10)},false);

</script>
</body>
</html>
```

بعد النقر على عنصر `<div>` فسيكون انتشار الحدث بالترتيب الآتي:

1. يُطلَق الحدث `click` في مرحلة capture على الكائن `window`
2. يُطلَق الحدث `click` في مرحلة capture على الكائن `document`
3. يُطلَق الحدث `click` في مرحلة capture على العنصر `<html>`
4. يُطلَق الحدث `click` في مرحلة capture على العنصر `<body>`
5. يُطلَق الحدث `click` في مرحلة capture على العنصر `<div>`
6. يُطلَق الحدث `click` في مرحلة bubble على العنصر `<div>`
7. يُطلَق الحدث `click` في مرحلة bubble على العنصر `<body>`
8. يُطلَق الحدث `click` في مرحلة bubble على العنصر `<html>`
9. يُطلَق الحدث `click` في مرحلة bubble على الكائن `document`
10. يُطلَق الحدث `click` في مرحلة bubble على الكائن `window`

استخدام مرحلة capture ليس شائعًا جدًا بسبب ضعف دعم المتصفحات لهذه المرحلة. لذا يُفترَض عادةً أنَّ الأحداث ستُطلَق في مرحلة bubble. سأحذف في الشيفرة الآتية كل ما يتعلق بمرحلة capture موضِّحًا ما الذي سيجري –عادةً– عند وقوع أحد الأحداث (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me to start event flow</div>

<script>

// bubble تقع ضمن مرحلة target مرحلة :1
document.querySelector('div').addEventListener('click',functi
on(){console.log(1);},false);

// bubble مرحلة :2
document.body.addEventListener('click',function()
{console.log(2);},false);

// bubble مرحلة :3
document.documentElement.addEventListener('click',function()
{console.log(3);},false);

// bubble مرحلة :4
document.addEventListener('click',function()
{console.log(4);},false);
```

```
// bubble مرحلة :5
window.addEventListener('click',function()
{console.log(5)},false);
</script>
</body>
</html>
```

لاحظ أنَّه لو نقرتَ على العنصر <body> في المثال السابق (أي في أي مكان ما عدا العنصر) فلن يُطلَق الحدث المرتبط بالعنصر وستبدأ مرحلة bubble من العنصر <body>، وذلك لأنَّ العنصر الهدف (event target) في هذه الحالة ليس العنصر وإنما <body>.

ملاحظات

- المتصفحات الحديثة تدعم استخدام مرحلة capture، فالأمر الذي كان يُعدّ غير عملي (لأنَّه قد يأتي أحدهم ويقاطع الحدث قبل أن يصل إلى العنصر الهدف) أصبح ذا فائدةٍ في هذه الآونة.

- أبقِ المعلومات السابقة التي تعلمتَها عن مرحلتَي capture و bubble في ذهنك عندما تقرأ عن «تفويض الأحداث» (event delegation) في هذا الفصل.

- يتضمن كائن الأحداث المُمرَّر إلى دوال معالجة الأحداث خاصيةً اسمها eventPhase التي تحتوي على عددٍ يُشير إلى المرحلة التي وقع الحدث فيها. فالقيمة 1 تُشير إلى مرحلة capture، والقيمة 2 تُشير إلى مرحلة الهدف (target)، والقيمة 3 تُشير إلى مرحلة bubble.

4. إضافة دوال معالجة أحداث إلى عقد العناصر والكائن Window و Document

الدالة `addEventListener()` متوافرة على جميع عقد العناصر (من النوع `Element`) وللكائنين `window` و `document` مما يعني إمكانية إضافة دوال معالجة أحداث إلى أجزاءٍ من مستند HTML بالإضافة إلى كائنات JavaScript التي تتعلق بشجرة DOM و BOM (اختصار إلى browser object model). سأستعمل هذه الدالة في المثال الآتي لأضيف الحدث `mousemove` إلى العنصر `<div>` وإلى الكائنين `document` و `window`. لاحظ -نتيجةً إلى انتشار الأحداث- أنَّ حركة الفأرة فوق العنصر `<div>` ستؤدي إلى استدعاء الدوال الثلاثة في كل مرة تحدث فيها حركة (مثال حي):

```
<!DOCTYPE html>
<html lang="en">

<body>

<div>mouse over me</div>

<script>

// إضافة الحدث mousemove إلى الكائن window
// مما يُطلِق الحدث ضمن مرحلة bubble
```

```
window.addEventListener('mousemove',function()
{console.log('moving over window');},false);

// إضافة الحدث mousemove إلى الكائن document
// مما يُطلِق الحدث ضمن مرحلة bubble
document.addEventListener('mousemove',function()
{console.log('moving over document');},false);

// إضافة الحدث mousemove إلى العنصر <div>
// مما يُطلِق الحدث ضمن مرحلة bubble
document.querySelector('div').addEventListener('mousemove',fu
nction(){console.log('moving over div');},false);

</script>
</body>
</html>
```

الدالة addEventListener() المُستخدَمة في الشيفرة السابقة تأخذ ثلاثة معاملات؛ أوّل معامل هو نوع الحدث الذي ستُنفَّذ الدالة عند وقوعه (يسمون ذلك بالمصطلح «استماع» [listen]). لاحظ أنَّ السلسلة النصية التي تُمثِّل نوع الحدث لا تحتوي على السابقة «on» (مثلًا: onmousemove). المعامل الثاني هو الدالة التي ستُستدعى عند وقوع الحدث. أما المعامل الثالث فهو قيمةٌ منطقيةٌ (Boolean) الغرض منها هو تحديد إن كان سيُطلَق الحدث في مرحلة capture أم لا.

ملاحظات

- تفاديتُ مناقشة طرائق معالجة الأحداث الأخرى (عبر خاصيات HTML و JavaScript) عمدًا لكي أحثّك على استعمال الدالة `addEventListener()`.

- عادةً، يرغب المطوِّرون في إطلاق الحدث أثناء مرحلة bubble لكي يتمكنوا العنصر من معالجة الحدث قبل أن ينتقل إلى بقية شجرة DOM. ولهذا السبب نضع قيمة المعامل الثالث المُمرَّر إلى الدالة `addEventListener()` في الغالبية العظمى من الحالات مساويةً إلى `false`. إن لم تُحدَّد قيمة المعامل الثالث في المتصفحات الحديثة فستُعدّ أنها `false`.

- حريٌ بك أن تعلم أنَّك تستطيع استخدام الدالة `addEventListener()` على الكائن `XMLHttpRequest`.

5. إزالة دوال معالجة الأحداث

يمكن أن تُستعمَل الدالة `removeEventListener()` لحذف الارتباط بين الأحداث ودوال معالجتها وذلك إن لم تكن دالة معالجة ذاك الحدث «دالةً مجهولةً» (anonymous function). سأضيف في الشيفرة الآتية دالتين لمعالجة الأحداث في مستند HTML وسأحاول إزالة ارتباطهما بحذفهما؛ لكن لن تُحذَف إلا الدالة غير المجهولة (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click to say hi</div>

<script>

var sayHi = function(){console.log('hi')};

// استعمال دالة مجهولة لمعالجة الحدث
document.body.addEventListener('click',function()
{console.log('dude');},false);

// استعمال دالة مُسنَدة إلى متغير لمعالجة الحدث
document.querySelector('div').addEventListener('click',sayHi,
false);

// محاولة حذف ربط الدالتين السابقتين بالحدث click
// لكن لن نستطيع حذف الدالة المجهولة
document.querySelector('div').removeEventListener('click',say
Hi,false);

// لا نستطيع فعل هذا لأنّ الدالة المُمرّرة
```

```
// removeEventListener إلى الدالة
// هي دالة جديدة مختلفة عن الدالة الأصلية
document.body.removeEventListener('click',function()
{console.log('dude');},false);

// سيؤدي الضغط على عنصر <div> إلى استدعاء الحدث المرتبط
// بالعنصر <body>، لعدم قدرتنا على حذفها

</script>
</body>
</html>
```

لا يمكن حذف الدوال المجهولة المستخدمة لمعالجة الأحداث والمُضافة عبر الدالة `addEventListener()`.

6. الحصول على خاصيات الكائن event

يُرسَل افتراضيًا إلى الدالة التي تُعالِج أحد الأحداث وسيطٌ يحتوي على جميع المعلومات المتعلقة بالحدث نفسه. سأشرح في المثال الآتي كيفية الوصول إلى الكائن `event` وعرض جميع خاصياته والقيم المرتبطة بها، وذلك للحدثين `load` و `click`. تذكر أنَّ عليك النقر فوق العنصر `<div>` لرؤية الخاصيات المرتبطة به (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>
document.querySelector('div').addEventListener('click',
function(event){
for(var p in event){
   // عرض خاصيات الكائن event والقيم المرتبطة بها
    console.log(p+' = '+event[p]);
}
},false);

// لا تنسَ أنّ this ستُشير إلى الكائن window
// عند استعمالها في المجال العام
this.addEventListener('load',function(event){
for(var p in event){
   // عرض خاصيات الكائن event والقيم المرتبطة بها
   console.log(p+' = '+event[p]);
}
}, false);
```

```
</script>
</body>
</html>
```

اعلم أنَّ كل حدث قد يحتوي على خاصيات مختلفة اعتمادًا على نوع الحدث (مثلًا: `MouseEvent` أو `KeyboardEvent` أو `WheelEvent`).

ملاحظات

- توجد في الكائن `event` الدوال `stopPropagation()` و `stopImediatePropagation()` و `preventDefault()`.

- سأستخدم في هذا الكتاب وسيطًا باسم `event` للإشارة إلى الكائن `event`، لكن في الحقيقة يمكنك استعمال أي اسم يحلو لك، ومن الشائع أن ترى وسيطًا باسم `e` أو `evt`.

7. قيمة this عند استعمال الدالة addEventListener()

قيمة الكلمة المحجوزة `this` داخل دوال معالجة الأحداث المُمرَّرة إلى `addEventListener()` ستُشير إلى العقدة أو الكائن الذي يرتبط الحدث به. سأضيف في المثال الآتي حدثًا إلى العنصر `<div>` ثم سأحاول الوصول باستعمال الكلمة المحجوزة `this` إلى العنصر `<div>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>
<script>

document.querySelector('div').addEventListener('click',
function(){
// ستُشير this إلى العنصر أو العقدة التي يرتبط بها الحدث
console.log(this); // '<div>'
}, false);

</script>
</body>
</html>
```

عندما تُطلَق الأحداث كجزءٍ من انتشار الأحداث (event flow)، فستبقى قيمة this مشيرةً إلى العقدة أو الكائن الذي يرتبط به الحدث. سأضيف في ما يلي الحدث click إلى العنصر <body>، ولو حاولتَ النقر على العنصر أو <body> فستبقى قيمة this مُشيرةً دومًا إلى <body> (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

// this وستبقى قيمة <body> أو <div> انقر على العنصر
// <body> مُشيرةً إلى عقدة العنصر
document.body.addEventListener('click',function(){
console.log(this); // <body>...</body>
},false);

</script>
</body>
</html>
```

إضافةً إلى ما سبق، من الممكن عبر استخدام الخاصية `event.currentTarget` الحصول على نفس المرجعية إلى العقدة أو الكائن المرتبط بالحدث التي توفرها الكلمة المحجوزة `this`. سأستعمل في المثال الآتي الخاصية `event.currentTarget` التابعة للكائن `event` لأريك أنَّها تحتوي على نفس قيمة `this` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.addEventListener('click',function(event){
console.log(event.currentTarget);  // '#document'
// نفس ناتج...
console.log(this);
}, false);

document.body.addEventListener('click',function(event){
console.log(event.currentTarget); // '<body>'
// نفس ناتج...
console.log(this);
}, false);

document.querySelector('div').addEventListener('click',functi
on(event){
console.log(event.currentTarget); // '<div>'
// نفس ناتج...
```

```
console.log(this);
}, false);
</script>
</body>
</html>
```

8. الإشارة إلى العنصر الهدف للحدث وليس العنصر الذي يرتبط به

بسبب وجود انتشار الأحداث، فمن الممكن النقر على عنصر موجودٍ ضمن العنصر <body>، مما يؤدي إلى استدعاء الدالة التي تُعالِج الحدث click على العنصر <body>. فعندما يحدث ذلك، فسيحتوي الكائن event المُمرَّر إلى دالة معالجة الحدث على خاصيةٍ باسم target ترتبط بالعقدة أو بالكائن الذي أدى إلى وقوع الحدث (أي ستُشير إلى العنصر «الهدف»).

عندما يُنقَر على العنصر في المثال الآتي، فستُستدعى دالة معالجة الحدث click المرتبط بالعنصر <body> وستُشير الخاصية event.target إلى العنصر الذي أدى إلى إطلاق الحدث. الخاصية event.target مفيدةٌ جدًا لكي نعلم منشأ الحدث (والذي لا نستطيع معرفته عبر استعمالنا للكلمة المحجوزة this) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<div>click me</div>

<script>

document.body.addEventListener('click',function(event){
// عندما ننقر على <div> فسيكون الناتج '<div>' لأنّ العنصر div
// هو «العنصر الهدف» الذي أدى إلى إطلاق الحدث
console.log(event.target);
}, false);

</script>
</body>
</html>
```

تخيل لو أننا نقرنا على العنصر <body> بدلًا من ، فستكون حينها قيمة الخاصية target وقيمة عقدة العنصر الذي أُطلق الحدث عليها هو نفس القيمة. أي أنَّ event.target و this و event.currentTarget ستشير جميعها إلى العنصر <body>.

9. تعطيل السلوك الافتراضي للأحداث باستخدام preventDefault()

توفِّر المتصفحات عدّة أحداث ترتبط افتراضيًا بعناصر مستند HTML المعروض للمستخدم. فمثلًا النقر على رابطٍ (link) سيؤدي إلى إطلاق حدث معيّن (مثلًا: الانتقال إلى الصفحة المعنية).

وكذلك الأمر بالنسبة إلى النقر على مربع اختيار أو الكتابة في حقل نصي (أي أنَّ النص سيظهر على الشاشة بعد كتابته). يمكن منع وقوع هذه الأحداث باستدعاء الدالة `preventDefault()` داخل دالة معالجة الحدث المرتبطة بعقد عنصر أو كائن. سأمنع في المثال الآتي الحدث الافتراضي من الوقوع في العناصر `<a>` و `<input>` و `<textarea>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<a href="google.com">no go</div>
<input type="checkbox" />
<textarea></textarea>
<script>

document.querySelector('a').addEventListener('click',function
(event){
// <a> منع السلوك الافتراضي للحدث المرتبط بالعنصر
// URL الذي هو تحميل رابط
event.preventDefault();
}, false);

document.querySelector('input').addEventListener('click',
function(event){
// checkbox منع السلوك الافتراضي للحدث المرتبط بعنصر الإدخال
```

```
// الذي هو تغيير حالة الاختيار للحقل
event.preventDefault();
},false);

document.querySelector('textarea')
          .addEventListener('keypress',function(event){
// منع السلوك الافتراضي للحدث المرتبط بعنصر textarea
// الذي هو إضافة الأحرف الجديدة المكتوبة
event.preventDefault();
},false);

//أبقِ في ذهنك أنّ تلك الأحداث ستنتشر كما رأينا سابقًا
// فالنقر على الرابط الموجود في مستند HTML الحالي لن يؤدي
// إلى إطلاق الحدث الافتراضي
// لكنه لن يمنع انتشار الحدث (عبر bubbling)
document.body.addEventListener('click', function(){
console.log('the event flow still flows!');
},false);
</script>
</body>
</html>
```

ستفشل جميع محاولات النقر على الرابط أو تغيير حالة مربع الاختيار أو كتابة نص في مربع النص السابق، لأننا نمنع وقوع الأحداث الافتراضية المرتبطة بتلك العناصر.

ملاحظات

- لا تمنع الدالة `preventDefault()` من انتشار الأحداث (عبر مرحلة capture أو bubble).

- وضع التعبير البرمجي `return false` في نهاية جسم الدالة التي تعالج الحدث يؤدي إلى نفس تأثير استدعاء الدالة `preventDefault()`.

- الكائن `event` المُمرَّر إلى دوال معالجة الأحداث يملك خاصيةً منطقيةً اسمها `cancelable` التي تُشير فيما إذا كان من الممكن إلغاء الحدث الافتراضي عبر استعمال الدالة `preventDefault()`.

- الكائن `event` المُمرَّر إلى دوال معالجة الأحداث يملك خاصيةً باسم `defaultPrevented` التي يكون لها القيمة `true` إن استعملنا الدالة `preventDefault()` على العنصر الرئيسي، وكان الحدث الحالي في مرحلة bubble.

10. إيقاف انتشار الأحداث

استدعاء الدالة `stopPropagation()` داخل الدالة التي تُعالِج الحدث سيؤدي إلى إيقاف انتشار الأحداث في مرحلتَي capture و bubble، لكن ستقع أيّة أحداث مرتبطة مباشرةً بالعقدة أو الكائن. لاحظ في المثال الآتي أنَّ الحدث `click` المرتبط بالعنصر `<body>` لن يقع أبدًا لأننا أوقفنا نشر الأحداث عبر شجرة DOM عند النقر على العنصر `<div>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.querySelector('div').addEventListener('click',
function(){
console.log('me too, but nothing from the event flow!');
}, false);

document.querySelector('div').addEventListener('click',
function(event){
console.log('invoked all click events attached, but cancel
capture and bubble event phases');
event.stopPropagation();
}, false);

document.querySelector('div').addEventListener('click',
function(){
console.log('me too, but nothing from the event flow!');
}, false);
```

```
// <div> لن يقع هذا الحدث عند النقر على العنصر
// <div> ذلك لأنّ الدالة التي تُعالِج الحدث click على العنصر
// توقف انتشار الأحداث
document.body.addEventListener('click', function(){
console.log('What, denied from being invoked!');
}, false);
</script>
</body>
</html>
```

لاحظ أنَّ أحداث النقر المرتبطة بالعنصر ستقع كما لو أنَّ شيئًا لم يتغير! أضف إلى ذلك أنَّ استخدام الدالة stopPropagation() لن يؤدي إلى إلغاء وقوع الأحداث الافتراضية، فلنقل أنَّ العنصر في المثال السابق كان عنصر <a>، ففي تلك الحالة لن يؤدي استخدام الدالة stopPropagation() إلى إيقاف السلوك الافتراضي للمتصفح (ألا وهو الانتقال إلى رابط URL المُعيّن).

11. إيقاف الأحداث وإيقاف نشر الأحداث على نفس العنصر

استدعاء الدالة stopImmediatePropagation() داخل دوال معالجة الأحداث سيؤدي إلى إيقاف نشر الأحداث (كالدالة stopPropagation()) إضافةً إلى إيقاف الأحداث المرتبطة مباشرةً بالعنصر الهدف والتي تُضاف إلى العنصر بعد تعريف دالة معالجة الأحداث التي تستدعي stopImmediatePropagation(). سأوضِّح ذلك في المثال الآتي الذي سأستدعي فيه الدالة

`stopImmediatePropagation()` في دالة معالجة الحدث الثاني المرتبط بالعنصر `<div>`، مما يؤدي إلى عدم استدعاء أيّة دوال مُعالِجَة للحدث `click` للعنصر `<div>` بعد ذلك (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

// ربط أوّل حدث
document.querySelector('div').addEventListener('click',
function(){
  console.log('I get invoked because I was attached first');
}, false);
// ربط ثاني حدث
document.querySelector('div').addEventListener('click',
function(event){
  console.log('I get invoked, but stop any other click events
on this target');
  event.stopImmediatePropagation();
```

```
}, false);

// ربط ثالث حدث، لكن بسبب استدعاء الدالة
// في الأعلى stopImmediatePropagation()
// فلن تُنفّذ دالة معالجة الحدث هذه
document.querySelector('div').addEventListener('click',
function(){
  console.log('I get stopped from the previous click event
listener');
}, false);

// لاحظ أنّ نشر الأحداث قد توقف كما لو أننا استدعينا الدالة
// أيضًا stopPropagation()
document.body.addEventListener('click',function(){
  console.log('What, denied from being invoked!');
}, false);
</script>
</body></html>
```

ملاحظة

استخدام الدالة stopImmediatePropagation() لن يؤدي إلى إلغاء الأحداث الافتراضية، إذ لا تلغى الأحداث الافتراضية إلا باستعمال الدالة preventDefault().

12. الأحداث المخصصة

لسنا محدودين بالأحداث المُعرَّفة مسبقًا، فمن الممكن إضافة حدث مخصص واستدعاؤه، وذلك باستخدام الدالة `addEventListener()` كالمعتاد بالإضافة إلى الدوال `document.createEvent()` و `initCustomEvent()` و `dispatchEvent()`.

سأُنشِئ في المثال الآتي حدثًا خاصًا اسمه `goBigBlue` وسأستدعي ذاك الحدث (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

var divElement = document.querySelector('div');


// إنشاء حدث مخصص، الحظ أنّ الوسيط CustomEvent مطلوبٌ
var cheer = document.createEvent('CustomEvent');

// إنشاء دالة لمعالجة وقوع الحدث الخاص
divElement.addEventListener('goBigBlue', function(event){
    console.log(event.detail.goBigBlueIs)
```

```
}, false);

// استخدام الدالة initCustomEvent لضبط تفاصيل الحدث المخصص
// معاملات الدالة initCustomEvent هي
// (الحدث، هل هو bubble؟، هل يمكن إلغاؤه؟
// هل نريد تمرير أيّة قيمة إلى event.detail؟)
cheer.initCustomEvent('goBigBlue',true, false,
{goBigBlueIs:'its gone!'});

// استدعاء الحدث المخصص باستخدام dispatchEvent
divElement.dispatchEvent(cheer);

</script>
</body>
</html>
```

ملاحظات

- يتطلب متصفح IE9 وجود معامل رابع إجباري للدالة `initiCustomEvent()`.

- أضافت مواصفة DOM 4 دالةً بانيةً باسم `CustomEvent()` التي تُبسِّط من إنشاء أحداث مخصص، لكنها غير مدعومة في متصفح IE9.

13. محاكاة أحداث الفأرة

لا تختلف محاكاة الأحداث عن إنشاء حدث مخصص. ففي حال أردنا محاكاة أحداث الفأرة فسنُنشِئ حدثًا باسم `MouseEvent` عبر الدالة `document.createEvent()` ثم سنضبط عبر الدالة `initMouseEvent()` أحداث الفأرة التي ستقع، ثم سنستدعيه على العنصر الذي نريد محاكاة الحدث عليه (مثلًا: العنصر `<div>` في مستند HTML).

يرتبط الحدث `click` في المثال الآتي بعنصر `<div>` الموجود في الصفحة؛ وبدلًا من النقر على العنصر `<div>` لكي يقع الحدث `click`، فسنحاكي ذلك برمجيًا بتهيئة حدث للفأرة ثم تنفيذه على العنصر `<div>` (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>
<div>no need to click, we programatically trigger it</div>
<script>

var divElement = document.querySelector('div');

// إعداد الحدث click الذي ستتم محاكاته
divElement.addEventListener('click', function(event){
    console.log(Object.keys(event));
}, false);
```

```
// إنشاء حدث لمحاكاة الحدث click
var simulateDivClick = document.createEvent('MouseEvents');

/* ضبط حركة الفأرة
initMouseEvent(type, bubbles, cancelable, view, detail,
screenx, screeny, clientx, clienty, ctrlKey, altKey,
shiftKey, metaKey, button, relatedTarget)*/
simulateDivClick.initMouseEvent('click', true, true,
document.defaultView, 0, 0, 0, 0, 0, false, false, false, 0,
null, null);

// استدعاء المحاكاة على العنصر <div>
divElement.dispatchEvent(simulateDivClick);

</script>
</body>
</html>
```

ملاحظة

محاكاة أحداث الفأرة تعمل على جميع المتصفحات في وقت كتابة هذا الكتاب. قد تُمسي محاكاة بقية الأحداث أكثر تعقيدًا، لذا سيصبح ضروريًا استخدام مكتبة jQuery (عبر الدالة `jQuery.trigger()` مثلًا).

14. تفويض الأحداث

يمكن تبسيط تعريف «تفويض الأحداث» (event delegation) على أنه الاستخدام البرمجي لنشر الأحداث (event flow) لربط دالة وحيدة لمعالجة عدد كبير من العناصر. أحد التأثيرات الجانبية لاستخدام تفويض الأحداث هو أنَّه ليس من الضروري أن تكون العناصر الهدف موجودةً في شجرة DOM عند إنشاء الحدث لكي تستطيع الدالةُ الاستجابةَ إلى الأحداث الواقعة على تلك العناصر. سنستفيد من ذلك عند التعامل مع ردود XHR التي تؤدي إلى تحديث شجرة DOM. وبالتالي ستتمكن العناصر الجديدة المُضافة إلى شجرة DOM بعد تفسير شيفرات JavaScript من الاستجابة مباشرةً إلى الأحداث. تخيل مثلًا جدولًا بعددٍ غير محدود من الأسطر والأعمدة. يمكننا عبر استعمال تفويض الأحداث من إضافة دالةٍ وحيدةٍ مرتبطةٍ بالعقدة `<table>`، وستُعدّ هذه العقدة على أنها العنصر الهدف الأولي للحدث.

النقر على أيّ عنصر `<td>` في الجدول الموجود في المثال الآتي سيؤدي إلى تفويض الحدث إلى الدالة التي تُعالِج الحدث `click` في العنصر `<table>`. لا تنسَ أنَّ كل هذا أصبح ممكنًا بفضل انتشار الأحداث (خصيصًا مرحلة bubble) (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Click a table cell</p>
```

```
<table border="1">
    <tbody>
        <tr><td>row 1 column 1</td><td>row 1 column
2</td></tr>
        <tr><td>row 2 column 1</td><td>row 2 column
2</td></tr>
        <tr><td>row 3 column 1</td><td>row 3 column
2</td></tr>
        <tr><td>row 4 column 1</td><td>row 4 column
2</td></tr>
        <tr><td>row 5 column 1</td><td>row 5 column
2</td></tr>
        <tr><td>row 6 column 1</td><td>row 6 column
2</td></tr>
    </tbody>
</table>
<script>

document.querySelector('table').addEventListener('click',func
tion(event){
    // td التأكد أنّ هذه الشيفرة ستُنفّذ فقط إذا كان الهدف هو
    if(event.target.tagName.toLowerCase() === 'td'){
        // td استخدام event.target للوصل إلى العنصر الهدف
         console.log(event.target.textContent);
```

```
    }
},false);

</script>
</body>
</html>
```

إذا حدَّثنا الجدول في المثال السابق وأضفنا أسطر جديدة إليه، فيجب أن تستجيب الأسطر الجديدة إلى الحدث `click` بعد إضافتها إلى الشاشة فورًا وذلك لأنَّ معالجة الحدث `click` ستفوَّض إلى العنصر `<table>`.

ملاحظة	من المنطقي استعمال تفويض الأحداث عندما تتعامل مع الأحداث `click` و `mousedown` و `mouseup` و `keydown` و `keyup` و `keypress`.

الفصل الثاني عشر:

إنشاء مكتبة للتعامل مع DOM

1. لمحة عن مكتبة dom.js

أريد منك أن تأخذ المعلومات التي حصلتَ عليها من هذا الكتاب، وتستعملها لإنشاء مكتبة صغيرة وحديثة وتُشبه في بنيتها jQuery الغرض منها هو التعامل مع DOM. تخيل أنَّ هذه المكتبة –التي سأدعوها dom.js– هي أساسٌ لمكتبةٍ حديثةٍ تُستعمَل لتحديد عقد DOM لتفعل شيئًا معها. ستوفر مكتبة dom.js إمكانية تحديد عناصر DOM (أو إنشائها) وإجراء عمليات عليها، بشكلٍ يشبه مكتبة jQuery كثيرًا. سأعرض بعض الأمثلة عن استخدام الدالة `dom()` التي يجب أن تبدو مألوفةً لديك إذا كانت عندك خبرةٌ مع مكتبة jQuery (أو غيرها من المكتبات التي تُستعمَل لتحديد العناصر):

```
// تحديد جميع عناصر li الموجودة ضمن ul والحصول
// على قيمة الخاصية innerHTML لأوّل عنصر li
dom('li','ul').html();

// إنشاء قطعة من المستند (document fragment) والحصول على
// خاصية innerHTML للعنصر ul الموجود داخلها
dom('<ul><li>hi</li></ul>').html()
```

يُمثِّل هذا الفصل لأغلبية القراء مثالًا عمليًا لتوظيف المعلومات التي أخذناها في هذا الكتاب وتطبيقها على مكتبة JavaScript للتعامل مع DOM. وقد يضفي هذا الفصل بعض الضوء على كيفية عمل مكتبة jQuery نفسها وكيفية إجراء عمليات معالجة DOM في أطر عمل JavaScript

المتـوافرة حاليًـا. أرجـو أن يُلهِم هـذا الفصـل القـراء ليكتبـوا مكتبـات أو أدوات للتعامـل مـع DOM عند الحاجة.

2. إنشاء مجال خاص

لحماية شيفرات مكتبة dom.js من المجال العام (global scope) فسنحتاج إلى إنشاء مجالٍ فريدٍ الذي نستطيع أن نجري فيه العمليات التي نريدها دون الخوف من حدوث تضاربات بين دوال ومتغيرات مكتبتنا وبين الدوال والمتغيرات الموجودة في المجال العام. سأضبط في الشيفرة الآتيـة دالـةً ذاتيـة الاسـتدعاء (Immediately-invoked function expression) لإنشاء المجال الخاص. فعند تنفيذ الدالة السابقة ستُضبَط قيمة المتغير `global` إلى الكائن العام الحالي (ألا وهو الكائن `window`) (الشيفرة):

```
(function(win){

var global = win;
var doc = this.document;

}}(window);
```

أنشأنا داخل الدالة السابقة متغيرين يشيران إلى الكائنين `window` و `document` (عبر المتغير `doc`) لتسريع الوصول إلى تلك الكائنات داخل الدالة.

3. إنشاء الدالة ()dom و ()GetOrMakeDom وإتاحتها إلى المجال العام

وكما في مكتبة jQuery، سنُنشِئ دالةً التي ستُعيد مجموعةً مغلفةً من عقد DOM (المجموعة المغلفة [wrapped set] هي مصفوفة خاصة شبيهة بالكائنات) (مثلًا: {0:ELEMENT_NODE, 1:ELEMENT_NODE, length:2}) اعتمادًا على محتوى المعاملات المُرسَلة إلى الدالة.

سأضبط في الشيفرة الآتية الدالة dom() ومعاملاتها التي ستُمرَّر إلى الدالة البانية GetOrMakeDOM() التي عندما تُستدعى فستُعيد كائنًا يحتوي على عقد DOM، والذي سيُعاد بدوره عبر الدالة dom() (الشيفرة):

```
(function(win){

var global = win;
var doc = global.document;

var dom = function(params,context){
    return new GetOrMakeDom(params,context);
};

var GetOrMakeDom = function(params,context){
};
```

```
})(window);
```

ولكي نستطيع الوصول إلى الدالة dom() من خارج المجال الخاص المُنشَأ عبر الدالة ذاتية الاستدعاء، فعلينا أن نجعل الدالة dom() متاحةً للمجال العام (أي عبر إنشاء مرجعية تُشير إليها). يمكن فعل ذلك بإنشاء خاصية (property) في المجال العام باسم dom وجعلها تُشير إلى الدالة المحلية dom(). وعندما سنحاول الوصول إلى الخاصية dom من المجال العام فستُشير إلى الدالة dom() الموجودة في المجال المحلي. سنفعل ذلك بإضافة التعبير البرمجي global.dom = dom; إلى الشيفرة السابقة (الشيفرة):

```
(function(win){

var global = win;
var doc = global.document;

var dom = function(params,context){
    return new GetOrMakeDom(params,context);
};

var GetOrMakeDom = function(params,context){

};
```

```
// جعل الدالة dom متاحةً في المجال العام
global.dom = dom;

})(window);
```

شيءٌ آخر علينا فعله هو إتاحة الوصول إلى الخاصية `GetOrMakeDom.prototype` في المجال العام. وكما في مكتبة jQuery (أقصد هنا الخاصية `jQuery.fn`) فسنوفِّر اختصارًا (ألا وهو `dom.fn`) يُشير إلى `GetOrMakeDom.prototype`، كما هو موضَّح في الشيفرة أدناه:

```
(function(win){

var global = win;
var doc = global.document;

var dom = function(params,context){
    return new GetOrMakeDom(params,context);
};

var GetOrMakeDom = function(params,context){
};

// جعل الدالة dom متاحةً في المجال العام
global.dom = dom;
```

```
// إنشاء اختصار إلى الخاصية prototype
dom.fn = GetOrMakeDom.prototype;

})(window);
```

عند إضافة أيّة دالة أو خاصية إلى dom.fn فستُضاف إلى الكائن GetOrMakeDOM.prototype وبالتالي ستتم وراثتها أثناء عملية البحث في سلسلة prototype في أيّة كائنات مُنشَأة من الدالة البانية GetOrMakeDOM().

ملاحظ: الدالة GetOrMakeDOM() ستُستدعى عبر المعامل new. احرص على فهم ما الذي يحدث عندما تُستدعى إحدى الدوال عبر المعامل new.

4. إضافة معامل اختياري لتحديد السياق في الدالة dom()

عند استدعاء الدالة dom() فستستدعي بدورها الدالة GetOrMakeDom() وتُمرِّر إليها جميع المعاملات التي أُرسِلَت إلى الدالة dom(). وعندما تُستدعى الدالة البانية GetOrMakeDom() فأوّل شيءٍ علينا تحديده هو السياق (context). يمكن ضبط السياق عند التعامل مع DOM عبر تمرير مرجعية إلى العقدة التي تريد البحث فيها. إن لم يكن الشرح السابق واضحًا فأحب أن أوضِّح أنَّ تمرير سياق إلى الدالة dom() سيوفِّر لنا القدرة على البحث عن عناصر DOM في «فرعٍ» معيّن في شجرة DOM. وهذا يشبه كثيرًا الوسيط الثاني الذي نُمرِّره إلى الدالة jQuery.

سأجعل السياق الافتراضي هو المستند الحالي الموجود في المجال العام، وإن كان معامل السياق متوافرًا، فسأُحدِّد ما هو (سلسلة نصية أم عقدة) وسأمرِّر العقدة كسياق للتحديد أو سأحدِّد العقدة عبر الدالة `querySelectorAll()` (إذا كان السياق عبارة عن سلسلة نصية) (الشيفرة):

```
(function(win){

var global = win;
var doc = global.document;

var dom = function(params,context){
    return new GetOrMakeDom(params,context);
};

var GetOrMakeDom = function(params,context){

    var currentContext = doc;
          if(context){
             // إما عقدة من النوع document أو النوع element
               if(context.nodeType){
                     currentContext = context;
                // أو أنها سلسلة نصية
                // فسنستعملها لتحديد العقدة
                }else{
```

```
                    currentContext =
doc.querySelector(context);
            }
        }

};

// جعل الدالة dom متاحةً في المجال العام
global.dom = dom;

// إنشاء اختصار إلى الخاصية prototype
dom.fn = GetOrMakeDom.prototype;
})(window);
```

بعد إضافة البنية المنطقية للمعامل `context` فأصبح بمقدورنا إضافة الشيفرة اللازمة للتعامل مع المعامل `params` الذي يُستخدَم لتحديد أو إنشاء العقد.

5. ملء وإعادة الكائن بمرجعيات إلى عقدة DOM المُحدَّدة اعتمادًا على المعامل params

يختلف المعامل `params` المُمرَّر إلى الدالة `dom()` ثم الذي سيُمرَّر إلى الدالة البانية `GetOrMakeDom()` بنوع القيمة التي قد يحملها. وهو يشبه أنواع القيم التي يمكن تمريرها إلى دالة `jQuery`:

- مُحدِّد CSS (مثلًا: `dom('body')`)
- سلسلة نصية تحتوي شيفرة HTML (مثلًا:
  `dom('<p>Hellow</p><p> World!</p>')`)
- عقدة من النوع `Element` (مثلًا: `dom(document.body)`)
- مصفوفة من عقد العناصر (مثلًا: `dom([document.body])`)
- قائمة من النوع `NodeList` (مثلًا: dom(document.body.children))
- كائن مُعاد من الدالة `dom()` نفسها (مثلًا: `dom(dom())`)

سينتج عند تمرير المعامل `params` إنشاء كائن يحتوي على مرجعيات إلى العقد (مثلًا: `{0:ELEMENT_NODE,1:ELEMENT_NODE,length:2}`) الموجودة في شجرة DOM أو في قطعة مستند (document fragment). لنتفحص عن قرب كل معامل من المعاملات السابقة الذي يمكن أن يُستعمَل لإنشاء كائن يحتوي على مرجعيات للعقد الناتجة.

البنية المنطقية اللازمة للسماح بتمرير مختلف الأنواع قيمةً لذاك المعامل معروضةٌ في الشيفرة التالية وتبدأ بتحققٍ بسيطٍ للتأكد أنَّ قيمة المعامل `params` ليست `undefined` وليست سلسلةً نصيةً فارغةً وليست سلسلةً نصية لا تحتوي إلى على الفراغات. وفي هذه الحالة سنُسند القيمة `0` إلى الخاصية `length` إلى الكائن المُنشَأ عبر استخدام الدالة `GetOrMakeDom()` وإعادة ذاك الكائن مما يؤدي إلى إنهاء تنفيذ الدالة. لو لم يكن المعامل `params` مساويًا للقيمة `false` (أو ما شابهها من القيم) فسيستمر تنفيذ الدالة.

إذا كانت قيمة المعامل `params` سلسلةً نصيةً فسنتحقق إن كانت تحتوي على شيفرة HTML. فلو احتوت السلسلة النصية على شيفرة HTML فسنُنشِئ قطعة مستند (document fragment) وستُستخدَم تلك السلسلة قيمةً للخاصية `innerHTML` للعنصر `<div>` الموجود في قطعة المستند لكي تُحوَّل السلسلة النصية إلى بُنية DOM. بعد تحويل السلسلة النصية التي تحتوي شيفرة HTML إلى شجرةٍ من العقد، فسنحاول الوصول إلى العقد الرئيسية فيها، ونضع مرجعيات إلى تلك العقد في الكائن المُنشَأ من الدالة `GetOrMakeDom()`، وإذا لم تحتوِ السلسلة النصية على شيفرة HTML فسيستمر تنفيذ الدالة.

سنتحقق الآن إذا كانت قيمة المعامل `params` هي مرجعيةٌ إلى عقدةٍ ما، وإذا كانت كذلك فسنضع تلك المرجعية في الكائن المُعاد من الدالة البانية وسنُعيده. أما لو لم تكن القيمة مرجعيةً فهذا يعني أنَّ قيمة `params` هي مصفوفةٌ أو قائمةٌ من النوع `HTMLCollection` أو `NodeList` أو سلسلة نصية تحتوي مُحدِّدًا (selector) أو كائن مُنشَأ من الدالة `dom()`. فلو كانت القيمة هي سلسلة نصية تحتوي مُحدِّدًا، فيمكن إنشاء قائمة بالعقد عبر استدعاء الدالة `querySelectorAll()` مع تمرير السياق (`currentContext`) إليها. إذا لم تكن قيمة المعامل سلسلةً نصيةً، فسنمر بحلقة تكرار على المصفوفة أو القائمة أو القائمة (بنوعيها) ونستخرج منها المرجعيات إلى العقد ثم نستعمل تلك المرجعيات كقيمة داخل الكائن الذي سيُعاد عند استدعاء الدالة البانية `GetOrMakeDom()`.

قد تشعر أنَّ البنية المنطقية للدالة `GetOrMakeDom()` معقدة وصعبة الفهم، لكن ضع ببالك أنَّ الغرض من الدالة البانية هو إنشاء كائن يحتوي على مرجعيات للعقد (مثلًا:

{0:ELEMENT_NODE,1:ELEMENT_NODE,length:2}) وإعـــادة ذلـــك الكـــائن إلى الدالـــة dom() (الشيفرة):

```
(function(win){

var global = win;
var doc = global.document;

var dom = function(params,context){
    return new GetOrMakeDom(params,context);
};

var regXContainsTag = /^\s*<(\w+|!)[^>]*>/;

var GetOrMakeDom = function(params,context){

    var currentContext = doc;
    if(context){
          if(context.nodeType){
                currentContext = context;
          }else{
                currentContext = doc.querySelector(context);
          }
    }
```

```
    // إذا لم يكن المعامل params مُعرّفًا، فأعد كائن dom() فارغ
    if(!params || params === '' || typeof params === 'string'
&& params.trim() === ''){
        this.length = 0;
        return this;
    }
    // إذا كان سلسلةً نصيةً، فأنشِئ قطعة مستند
    // واملأ ذاك الكائن واعده من الدالة
    if(typeof params === 'string' &&
regXContainsTag.test(params)){
        // قيمة المعامل تُمثِّل شيفرة HTML
        // لنُنشِئ عنصر div وقطعة مستند، ثم نضيف div إلى
        // div ثم نضبط خاصية innerHTML للعنصر docfrag
        // إلى السلسلة النصية
        // ثم نحصل على مرجعية لأوّل عنصر ابن
        var divElm = currentContext.createElement('div');
        divElm.className = 'hippo-doc-frag-wrapper';
        var docFrag =
          currentContext.createDocumentFragment();
        docFrag.appendChild(divElm);
        var queryDiv = docFrag.querySelector('div');
        queryDiv.innerHTML = params;
        var numberOfChildren = queryDiv.children.length;
```

```
        // المرور على عناصر nodelist وملء الكائن
        // سنحتاج إلى ذلك لأنّ سلسلة HTML المُمرّرة قد
        // تحتوي على عدِّة عناصر «أخوة» (siblings)
        for (var z = 0; z < numberOfChildren; z++) {
              this[z] = queryDiv.children[z];
        }
        // إعطاء قيمة للخاصية length التابعة للكائن
        this.length = numberOfChildren;
        // إعادة الكائن
        return this;
        // مثلًا {0:ELEMENT_NODE,1:ELEMENT_NODE,length:2}
}

// إذا مُرِّرَت مرجعية إلى عقدة وحيدة، فاملأ الكائن ثم أعده
if(typeof params === 'object' && params.nodeName){
        this.length = 1;
        this[0] = params;
        return this;
}

 // إذا كان المعامل params كائنًا، لكنه ليس عقدةً
 // فإما أن يكون NodeList أو مصفوفة
 // أو أن يكون مُحدِّدًا نصيًا
 // لذا لنُنشِئ متغيرًا من النوع NodeList
```

```
    var nodes;
    if(typeof params !== 'string'){
        // إما أن يكون مصفوفةً أو قائمةً من النوع NodeList
        nodes = params;
    }else{
        // حسنًا، إنه سلسلةٌ نصية
        nodes =
          currentContext.querySelectorAll(params.trim());
    }

    // المرور على المصفوفة أو قائمة NodeList التي
    // أنشأناها أعلاه وملء الكائن بمحتوياتها
    var nodeLength = nodes.length;
    for (var i = 0; i < nodeLength; i++) {
        this[i] = nodes[i];
    }
    // إعطاء الكائن قيمةً للخاصية length
    this.length = nodeLength;

    // إعادة الكائن
    return this;
    // مثلًا {0:ELEMENT_NODE,1:ELEMENT_NODE,length:2}

};
```

```
// جعل الدالة dom متاحةً في المجال العام
global.dom = dom;

// إنشاء اختصار إلى الخاصية prototype
dom.fn = GetOrMakeDom.prototype;

})(window);
```

6. إنشاء الدالة ()each

عندما نستدعي الدالة ()dom فسنستطيع الوصول إلى أي شيء مرتبط بالخاصية doc.fn وذلك بفضل الوراثة عبر سلسلة prototype (مثلًا: ()dom().each)؛ وبآلية شبيهةٍ بالآلية المستعملة في مكتبة jQuery، ستُجري الدوال المرتبطة بالخاصية dom.fn عملياتها على الكائنات المُنشَأة من الدالة البانية ()GetOrMakeDom.

ما يلي هو الشيفرة المسؤولة عن الدالة ()each (الشيفرة):

```
dom.fn.each = function (callback) {

    // الكائن الحالي الذي نعمل عليه مُنشَأٌ من
    // الدالة البانية ()GetOrMakeDom
    // ومُعادٌ بعد استدعاء ()dom
```

```
    var len = this.length;
    for(var i = 0; i < len; i++){
        // استدعاء الدالة التي ستُنفّذ على كل عنصر من العناصر
        callback.call(this[i], i, this[i]);
    }
};
```

وكما توقعت، تأخذ الدالة `each()` وسيطًا هو الدالة التي ستُستدعى على كل عنصر من العناصر المُحدَّدة والموجودة في الكائن المُنشَأ من الدالة البانية `GetOrMakeDom()`. القيمة `this` الموجودة داخل الدالة `each()` تُشير إلى نسخة الكائن المُنشَأة من الدالة `GetOrMakeDom()` (مثلًا: `{0:ELEMENT_NODE,1:ELEMENT_NODE,length:2}`).

عندما لا تُعيد الدالة قيمةً (لو نفذنا مثلًا `dom().length` فسنحصل على قيمةٍ هي عدد العناصر في الكائن) فمن الممكن السماح بإنشاء «سلسلة» (chain) من الدوال بإعادة الكائن مرةً أخرى بدلًا من إعادة قيمة مُحدَّدة. أي أننا سنُعيد الكائن المُنشأ من `GetOrMakeDom()` لكي تستطيع دالةٌ أخرى التعامل معه. أردتُ أن تصبح الدالة `each()` قابلةً لأن تكون جزءًا من سلسلةٍ من الدوال، أي يمكن استدعاء دوال أخرى بعد الدالة `each()`، لذا سأُعيد في نهاية الدالة القيمة `this`. وأذكِّرك أنَّ قيمة `this` في الشيفرة الآتية تُمثِّل نسخة الكائن المُنشَأة عند استدعاء الدالة البانية `GetOrMakeDom()` (الشيفرة):

```
dom.fn.each = function (callback) {
    var len = this.length;
    for(var i = 0; i < len; i++){
          callback.call(this[i], i, this[i]);
    }
    // هذا سيجعل من الممكن وضع الدالة each في سلسلة
    return this;
};
```

7. إنشاء الدوال html() و append() و text()

بعد أن أنشأنا الدالة الرئيسية `each()` وجعلنا الدوران الضمني (implicit iteration) متاحًا لنا، فيمكننا الآن بناء بضع دوال التي تستطيع إجراء عمليات على العقد التي نُحدِّدها من مستند HTML أو نُنشِئها في قطع المستندات. هذه هي الدوال الثلاث التي سنُنشئها:

- `html() / html('html string')`
- `text() / text('text string')`
- `append('html | text | dom() | nodelist/HTML collection | node | array')`

الدالتان `html()` و `text()` لهما نفس البنية تقريبًا، فلو استدعيت الدالة بعد تمرير معامل إليها فسنمرّ (باستخدام `dom.fn.each()` للدوران الضمني) على كل عقدة عنصر موجودة في نسخة الكائن المُنشَأة من `GetOrMakeDom()` وسنضبط قيمة الخاصية `innerHTML` أو `textContent`. إذا لم يُمرَّر أي متغير إليهما فسنُعيد قيمة الخاصية `innerHTML` أو

textContent لأوّل عقدة عنصر في الكائن المُنشَأ من GetOrMakeDom() (الشيفرة):

```
dom.fn.html = function(htmlString){
    if(htmlString){
        return this.each(function(){
            // لاحظ أنني سأعيد القيمة this مما يمكّننا
            // من استخدام هذه الدالة في سلسلة
             this.innerHTML = htmlString;
        });
    }else{
        return this[0].innerHTML;
    }
};

dom.fn.text = function(textString){
    if(textString){
        return this.each(function(){
            this.textContent = textString;
        });
    }else{
        return this[0].textContent.trim();
    }
};
```

الدالة append() ستستفيد من الدالة insertAdjacentHTML()، وتقبل تمرير سلسلة تحوي شيفرة HTML أو سلسلة نصية أو كائن dom() أو قائمة من النوع NodeList أو HTMLCollection أو عقدة وحيدة أو مصفوفة فيها عدّة عقد، وتُضيفها إلى العقد المُحدَّدة (الشيفرة):

```
dom.fn.append = function(stringOrObject){
    return this.each(function(){
          if(typeof stringOrObject === 'string'){

    this.insertAdjacentHTML('beforeend',stringOrObject);
          }else{
                var that = this;
                dom(stringOrObject).each(function(name,value)
{

    that.insertAdjacentHTML('beforeend',value.outerHTML);
                });
          }
    });
};
```

8. تجربة مكتبة dom.js

أثناء فترة إنشائي لمكتبة dom.js، أنشأتُ معها بعض اختبارات qunit التي سنُشغِّلها الآن خارج نطاق إطار عمل الاختبار. لكن يمكنك أيضًا تشغيل إطار عمل الاختبار لرؤية مكتبة dom.js عمليًا. سيوضِّح هذا المثال كيفية عمل المكتبة التي أنشأناها في هذا الفصل (مثال حي):

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li>1</li>
<li>2</li>
<li>3</li>
</ul>

<script src="https://raw.github.com/codylindley/domjs/master/
builds/dom.js"></script>
<script>

//dom()
console.log(dom());
console.log(dom(''));
console.log(dom('body'));
```

```
console.log(dom('<p>Hellow</p><p> World!</p>'));
console.log(dom(document.body));
console.log(dom([document.body, document.body]));
console.log(dom(document.body.children));
console.log(dom(dom('body')));

//dom().html()
console.log(dom('ul li:first-child').html('one'));
console.log(dom('ul li:first-child').html() === 'one');

//dom().text()
console.log(dom('ul li:last-child').text('three'));
console.log(dom('ul li:last-child').text() === 'three');

//dom().append()
dom('ul').append('<li>4</li>');
dom('ul').append(document.createElement('li'));
dom('ul').append(dom('li:first-child'));

</script>
</body>
</html>
```

9. الخلاصة

كان هذا الفصل عن إنشاء مكتبة بسيطة للتعامل مع DOM شبيهة بمكتبة jQuery. إذا أردتَ أن تكمل بدراسة الأجزاء المكونة لمكتبات شبيهة بمكتبة jQuery فأنصحك بالنظر إلى hippo.js، التي تُمثِّل تمرينًا لإعادة إنشاء دوال DOM الموجودة في jQuery للتعامل مع المتصفحات الحديثة. تستعمل مكتبتا dom.js و hippo.js أداة البناء grunt و QUnit للاختبار و JS Hint للمساعدة في اكتشاف الأخطاء، وأنا أحثُّك بشدة أن تنظر إليها إن كنتَ تفكر في إنشاء مكتبات JavaScript خاصة بك. أنصحك أيضًا أن تقرأ «Designing Better JavaScript APIs» قبل أن تنطلق لبناء تطبيقات تتعامل مع DOM. بالتوفيق!