البرمجة بلغة جافاسكربت

تأليف

Marijn Haverbeke

ترجمة

أسامة دمراني

أكاديمية حسوب

البرمجة بلغة جافاسكربت

مرجعك الأساسي إلى تعلم البرمجة وكتابة البرامج والتطبيقات بلغة جافاسكربت

اسم الكتاب: البرمجة بلغة جافاسكربت

المؤلف: مارين هافربيك

المترجم: أسامة دمراني

المحرر: جميل بيلوني، آيات اليطقان

تصميم الغلاف: محمد زاهر شلار

سنة النشر: 2022

رقم الإصدار: 1.0

Book Title: Eloquent JavaScript

Author: Marijn Haverbeke

Translator: Osama Damarany

Editor: Jamil Bailony, Ayat Alyatakan

Cover Design: Mohamed Zaher Shallar

Publication Year: 2022

Edition: 1.0

بعض الحقوق محفوظة - أكاديمية حسوب.

أكاديمية حسوب أحد مشاريع شركة حسوب محدودة المسؤولية.

مسجلة في المملكة المتحدة برقم 07571594.

Hsoub Limited

Level 17, Dashwood House

69 Old Broad Street

London EC2M 1QS

United Kingdom

https://academy.hsoub.com

academy@hsoub.com

Copyright Notice

The author publishes this work under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

This license is acceptable for Free Cultural Works.
The licensor cannot revoke these freedoms as long as you follow the license terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
Read the text of the full license on the following link:

إشعار حقوق التأليف والنشر

ينشر المصنِّف هذا العمل وفقا لرخصة المشاع الإبداعي نَسب المُصنَّف - غير تجاري - الترخيص بالمثل 4.0 دولي (CC BY-NC-SA 4.0).

لك مطلق الحرية في:

- المشاركة — نسخ وتوزيع ونقل العمل لأي وسط أو شكل.
- التعديل — المزج، التحويل، والإضافة على العمل.

هذه الرخصة متوافقة مع أعمال الثقافة الحرة.
لا يمكن للمرخِّص إلغاء هذه الصلاحيات طالما اتبعت شروط الرخصة:

- نَسب المُصنَّف — يجب عليك نَسب العمل لصاحبه بطريقة مناسبة، وتوفير رابط للترخيص، وبيان إذا ما قد أُجريت أي تعديلات على العمل. يمكنك القيام بهذا بأي طريقة مناسبة، ولكن على ألا يتم ذلك بطريقة توحي بأن المؤلف أو المرخِّص مؤيد لك أو لعملك.
- غير تجاري — لا يمكنك استخدام هذا العمل لأغراض تجارية.
- الترخيص بالمثل — إذا قمت بأي تعديل، تغيير، أو إضافة على هذا العمل، فيجب عليك توزيع العمل الناتج بنفس شروط ترخيص العمل الأصلي.

منع القيود الإضافية — يجب عليك ألا تطبق أي شروط قانونية أو تدابير تكنولوجية تقيد الآخرين من ممارسة الصلاحيات التي تسمح بها الرخصة.
اقرأ النص الكامل للرخصة عبر الرابط التالي:

https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode

The illustrations used in this book is created by the author and all are licensed with a license compatible with the previously stated license.

الصور المستخدمة في هذا الكتاب من إعداد المؤلف وهي كلها مرخصة برخصة متوافقة مع الرخصة السابقة.

عن الناشـر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى توفير دروس وكتب عالية الجودة في مختلـف المجـالات وتقـدم دورات شـاملة لتعلُّم البرمجــة بأحــدث تقنياتهــا معتمــدةً على التطــبيق العملي الــذي يؤهــل الطــالب لــدخول ســوق العمل بثقة.

حسوب مجموعة تقنية في مهمة لتطوير العالم العربي. تبني حسوب منتجـات تركِّز على تحسـين مسـتقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة تطوير التطبيقات
باستخدام لغة JavaScript

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

المحتويات باختصار

مقدمة 20

1. القيم والأنواع والعوامل 29

2. هيكل البرنامج 40

3. الدوال 58

4. هياكل البيانات: الكائنات والمصفوفات 78

5. الدوال العليا higher-order functions 106

6. الحياة السرية للكائنات 123

7. مشروع الروبوت 145

8. الزلات البرمجية والأخطاء 159

9. التعابير النمطية Regular Expressions 177

10. الوحدات Modules 203

11. البرمجة غيـر المتـزامنة 218

12. مشروع لغة برمجة 242

13. جافاسكربت والمتصفحات 259

14. نموذج كائن المستند DOM 267

15. معالجة الأحداث 289

16. مشروع لعبة منصة 309

17. الرسم على لوحة 336

18. طلبيات HTTP والاستمارات 361

19. مشروع محرر رسوم نقطية 384

20. بيئة Node.js: جافاسكربت خارج المتصفح 410

21. بناء موقع عبـر Node.js 429

جدول المحتويات

مقدمة **20**
عن البرمجة 21
سبب أهمية اللغات 22
ما هي جافاسكربت؟ 25
الشيفرة البـرمجية واستخداماتها 26
نظرة عامة على هذا الكتاب 27
الاصطلاحات المطبعية في الكتاب 28
المساهمة 28
1. القيم والأنواع والعوامل **29**
1.1 القيم 30
1.2 الأعداد 30
1.2.1 الحساب 31
1.2.2 الأعداد الخاصة 32
1.3 السلاسل النصية 32
1.4 العوامل الأحادية 34
1.5 القيم البوليانية 34
1.5.1 الموازنة 34
1.5.2 العوامل المنطقية 35
1.6 القيم الفارغة 37
1.7 التحويل التلقائي للنوع 37
1.7.1 اختصار العوامل ومقاطعتها 38
1.8 خاتمة 39
2. هيكل البرنامج **40**
2.1 التعابير والتعليمات البرمجية 40
2.2 الرابطة Binding 41
2.3 أسماء الرابطة 43
2.4 البيئة 43

2.5 الدوال 43
2.6 دالة console.log 44
2.7 القيم المعادة 45
2.8 تدفق التحكم 45
2.9 تنفيذ شرطي 46
2.10 حلقات while وdo 47
2.11 الشيفرة المزاحة 49
2.12 حلقات for 50
2.13 الهروب من حلقة 51
2.14 تحديث الرابطة بإيجاز 51
2.15 الإرسال إلى قيمة باستخدام التعليمة switch 52
2.16 الحالة الكبيرة للأحرف 53
2.17 التعليقات 53
2.18 خاتمة 54
2.19 تدريبات 54
2.19.1 مثلث التكرار 55
2.19.2 FizzBuzz 55
2.19.3 لوحة الشطرنج 56
3. الدوال 58
3.1 تعريف الدالة 58
3.2 الرابطات Bindings والنطاقات Scopes 60
3.2.1 النطاق المتشعب 61
3.3 الدوال على أساس قيم 62
3.4 مفهوم التصريح 62
3.5 الدوال السهمية Arrow Functions 63
3.6 مكدس الاستدعاء The call stack 64
3.7 الوسائط الاختيارية Optional arguments 65
3.8 التغليف Closure 66
3.9 التعاود Recursion 68

3.10 الدوال النامية Growing Functions 71
3.11 الدوال والآثار الجانبية 73
3.12 خاتمة 74
3.13 تدريبات 74
3.13.1 القيمة الصغرى 74
3.13.2 التعاود Recursion 75
3.13.3 عد حبات الفول 76
4. هياكل البيانات: الكائنات والمصفوفات 78
4.1 الإنسان المتحول إلى سنجاب 78
4.2 مجموعات البيانات 79
4.3 الخصائص 80
4.4 التوابع Methods 81
4.5 الكائنات Objects 82
4.6 قابلية التغير Mutability 84
4.7 سجل المستذئب 85
4.8 حساب علاقة الترابط 87
4.9 حلقات المصفوفات التكرارية 89
4.10 التحليل النهائي 89
4.11 زيادة على المصفوفات 91
4.12 السلاسل النصية وخصائصها 93
4.13 معامل الباقي rest 95
4.14 الكائن Math 95
4.15 التفكيك 97
4.16 صيغة JSON 98
4.17 خاتمة 99
4.18 تدريبات 99
4.18.1 مجموع مجال ما 99
4.18.2 عكس مصفوفة 101
4.18.3 قائمة 102

4.18.4 الموازنة العميقة 104
5. الدوال العليا higher-order functions 106
5.1 التجريد Abstractions 107
5.2 تجريد التكرار 108
5.3 الدوال العليا 109
5.4 مجموعات البيانات النصية 110
5.5 ترشيح المصفوفات 111
5.6 التحويل مع map 112
5.7 التلخيص باستخدام reduce 113
5.8 قابلية التركيب 114
5.9 السلاسل النصية ورموز المحارف 116
5.10 التعرف على النصوص 118
5.11 خاتمة 119
5.12 تدريبات 119
5.12.1 التبسيط 119
5.12.2 الحلقة التكرارية الخاصة بك 120
5.12.3 كل شيء 120
5.12.4 اتجاه الكتابة السائد 121
6. الحياة السرية للكائنات 123
6.1 التغليف Encapsulation 123
6.2 التوابع Methods 124
6.3 النماذج الأولية Prototypes 125
6.4 الأصناف Classes 127
6.5 صياغة الصنف Class Notation 128
6.6 إعادة تعريف الخصائص المشتقة 129
6.7 الخرائط Maps 130
6.8 تعددية الأشكال Polymorphism 132
6.9 الرموز Symbols 133
6.10 واجهة المكرر iterator 134

6.11 التوابع الجالبة والضابطة والساكنة 137
6.12 الوراثة Inheritance 138
6.13 عامل instanceof 140
6.14 خاتمة 140
6.15 تدريبات 141
6.15.1 النوع المتجهي 141
6.15.2 المجموعات 142
6.15.3 المجموعات القابلة للتكرار 143
6.15.4 استعارة تابع 144
7. مشروع الروبوت 145
7.1 قرية المرج Meadowfield 145
7.2 المهمة 147
7.3 البيانات الثابتة Persistent Data 149
7.4 المحاكاة 150
7.5 طريق شاحنة البريد 152
7.6 اكتشاف الطريق Pathfinding 153
7.7 تدريبات 155
7.7.1 معايرة الروبوت 155
7.7.2 كفاءة الروبوت 156
7.7.3 المجموعة الثابتة 156
8. الزلات البرمجية والأخطاء 159
8.1 اللغة 159
8.2 الوضع الصارم strict mode 160
8.3 الأنواع Types 161
8.4 الاختبار 162
8.5 التنقيح Debugging 163
8.6 توليد الخطأ 165
8.7 الاستثناءات Exceptions 166
8.8 التنظيف وراء الاستثناءات 168

8.9 الالتقاط الانتقائي 170
8.10 التوكيدات Assertions 172
8.11 خاتمة 173
8.12 تدريبات 173
8.12.1 Retry 173
8.12.2 الصندوق المغلق 174
9. التعابير النمطية Regular Expressions 177
9.1 إنشاء تعبير نمطي 177
9.2 التحقق من المطابقات 178
9.3 مجموعات المحارف 178
9.4 تكرار أجزاء من النمط 180
9.5 جمع التعبيرات الفرعية 181
9.6 التطابقات والمجموعات 181
9.7 صنف التاريخ 182
9.8 حدود الكلمة والسلسلة النصية 184
9.9 أنماط الاختيار 185
9.10 آلية المطابقة 185
9.11 التعقب الخلفي Backtracking 186
9.12 التابع replace 188
9.13 الجشع Greed 190
9.14 إنشاء كائنات RegExp ديناميكيا 191
9.15 التابع search 192
9.16 خاصية lastIndex 192
9.16.1 التكرار على التطابقات 193
9.17 تحليل ملف INI 194
9.18 المحارف الدولية 196
9.19 خاتمة 198
9.20 تدريبات 199
9.20.1 Regexp golf 199

9.20.2 أسلوب الاقتباس 201
9.20.3 الأعداد مرة أخرى 201
10. الوحدات Modules 203
10.1 الوحدات Modules 204
10.2 الحزم packages 204
10.3 الوحدات المرتجلة Improvised modules 206
10.4 تقييم البيانات على أساس شيفرات 206
10.5 CommonJS 207
10.6 وحدات ECMAScript 210
10.7 البناء والتجميع 211
10.8 تصميم الوحدة 212
10.9 خاتمة 214
10.10 تدريبات 214
10.10.1 الروبوت التركيبـي 214
10.10.2 وحدة الطرق 216
10.10.3 الاعتماد المتبادل بين الوحدات 217
11. البرمجة غيـر المتـزامنة 218
11.1 عدم التزامن Asynchronicity 218
11.2 تقنية الغراب 220
11.3 ردود النداء Callbacks 221
11.4 الوعود 223
11.5 الفشل Failure 224
11.6 صعوبة الشبكات 225
11.7 تجميعات الوعود 227
11.8 إغراق الشبكة Network flooding 228
11.9 توجيه الرسائل 229
11.10 دوال async 232
11.11 المولدات Generators 234
11.12 حلقة الحدث التكرارية 235

11.13 زلات البرامج غير المتزامنة 237
11.14 خاتمة 238
11.15 تدريبات 239
11.15.1 تتبع المشرط 239
11.15.2 بناء Promise.all 240
12. مشروع لغة برمجة 242
12.1 التحليل 242
12.2 التقييم 247
12.3 الصيغ الخاصة 248
12.4 البيئة 250
12.5 الدوال 251
12.6 التصـريف 253
12.7 التغليف 253
12.8 تدريبات 254
12.8.1 المصفوفات 254
12.8.2 التغليف Closure 255
12.8.3 التعليقات 256
12.8.4 إيجاد النطاق 257
13. جافاسكربت والمتصفحات 259
13.1 الشبكات والإنترنت 260
13.2 الشبكة العنكبوتية العالمية The Web 261
13.3 HTML 262
13.4 HTML وجافاسكربت 264
13.5 داخل صندوق الاختبارات sandbox 265
13.6 التوافقية وحروب المتصفحات 266
14. نموذج كائن المستند DOM 267
14.1 هيكل المستند 267
14.2 الأشجار 268
14.3 المعيار 270

14.4 التنقل داخل الشجرة 270
14.5 البحث عن العناصر 272
14.6 تغيير المستند 273
14.7 إنشاء العقد 273
14.8 السمات Attributes 275
14.9 مخطط المستند Layout 276
14.10 التنسيق Styling 278
14.11 التنسيقات الانسيابية Cascading Styles 279
14.12 محددات الاستعلامات Query Selectors 281
14.13 التموضع والتحريك 282
14.14 خاتمة 284
14.15 تدريبات 284
14.15.1 بناء جدول 284
14.15.2 جلب العناصر بأسماء وسومها 286
14.15.3 قبعة القطة 287
15. معالجة الأحداث 289
15.1 مفهوم معالجات الأحداث Events Handlers 289
15.2 الأحداث وعقد DOM 290
15.3 كائنات الأحداث 291
15.4 الانتشار Propagation 292
15.5 الإجراءات الافتراضية 293
15.6 أحداث المفاتيح 294
15.7 أحداث المؤشر 295
15.7.1 ضغطات الفأرة 295
15.7.2 حركة الفأرة 297
15.7.3 أحداث اللمس 298
15.8 أحداث التمرير 299
15.9 أحداث التنشيط Focus Events 300
15.10 حدث التحميل Load Event 301

15.11 الأحداث وحلقات الأحداث التكرارية 302
15.12 المؤقتات Timers 303
15.13 تقييد إطلاق الحدث Debouncing 304
15.14 خاتمة 305
15.15 تدريبات 305
15.15.1 بالون 305
15.15.2 ذيل الفأرة 306
15.15.3 التبويبات Tabs 307
16. مشروع لعبة منصة 309
16.1 اللعبة 309
16.2 التقنية 310
16.3 المستويات 311
16.4 قراءة المستوى 311
16.5 الكائنات الفاعلة Actors 313
16.6 مشكلة التغليف 317
16.7 الرسم 317
16.8 الحركة والتصادم 323
16.9 تحديثات الكائنات الفاعلة 326
16.10 مفاتيح التعقب 328
16.11 تشغيل اللعبة 329
16.12 تدريبات 331
16.12.1 انتهاء اللعبة 331
16.12.2 الإيقاف المؤقت للعبة 332
16.12.3 الوحش 334
17. الرسم على لوحة 336
17.1 الرسوميات المتجهية القابلة للتحجيم SVG 337
17.2 عنصر اللوحة 337
17.3 الأسطر والأسطح 338
17.4 المسارات 339

17.5 المنحنيات 340
17.6 رسم المخطط الدائري 342
17.7 النصوص 343
17.8 الصور 343
17.9 التحول 345
17.10 تخزين التحويلات ومحوها 347
17.11 عودة إلى اللعبة 349
17.12 اختيار واجهة الرسوميات 354
17.13 خاتمة 355
17.14 تدريبات 355
17.14.1 الأشكال 355
17.14.2 المخطط الدائري 357
17.14.3 الكرة المرتدة 358
17.14.4 الانعكاس المحسوب مسبقا 359
18. طلبيات HTTP والاستمارات 361
18.1 البروتوكول 361
18.2 المتصفحات وHTTP 363
18.3 واجهة Fetch 364
18.4 صندوق اختبارات HTTP 366
18.5 تقدير HTTP 366
18.6 الأمان وHTTP 367
18.7 حقول الاستمارات 367
18.8 التركيز Focus 369
18.9 الحقول المعطلة 370
18.10 الاستمارات على أساس عنصر كامل 370
18.11 الحقول النصية 372
18.12 أزرار الاختيار وأزرار الانتقاء 373
18.13 حقول التحديد 374
18.14 حقول الملفات 375

18.15 تخزين البيانات في جانب العميل 377
18.16 خاتمة 380
18.17 تدريبات 380
18.17.1 التفاوض على المحتوى 380
18.17.2 طاولة عمل جافاسكربت 381
18.17.3 لعبة حياة كونويل 382
19. مشروع محرر رسوم نقطية 384
19.1 المكونات 385
19.2 الحالة 386
19.3 بناء DOM 387
19.4 اللوحة Canvas 388
19.5 التطبيق 391
19.6 أدوات الرسم 393
19.7 الحفظ والتحميل 396
19.8 سجل التغييرات Undo History 399
19.9 لنرسم 401
19.10 سبب صعوبة البرنامج 402
19.11 تدريبات 403
19.11.1 رابطات لوحة المفاتيح 403
19.11.2 الرسم بكفاءة 404
19.11.3 الدوائر 406
19.11.4 الخطوط المستقيمة 407
20. بيئة Node.js: جافاسكربت خارج المتصفح 410
20.1 تقديم إلى Node 410
20.2 الأمر node 411
20.3 الوحدات Modules 412
20.4 التثبيت باستخدام NPM 413
20.5 ملفات الحزم 414
20.6 الإصدارات 415

20.7 وحدة نظام الملفات 415
20.8 وحدة HTTP 417
20.9 البث Stream 419
20.10 خادم الملفات 420
20.11 خاتمة 426
20.12 تدريبات 426
20.12.1 أداة بحث 426
20.12.2 إنشاء المجلد 427
20.12.3 مساحة عامة على الويب 427
21. بناء موقع عبـر Node.js 429
21.1 التصميم 429
21.2 الاستطلاع المفتوح 430
21.3 واجهة HTTP 431
21.4 الخادم 433
21.4.1 التوجيه Routing 433
21.4.2 تخديم الملفات 434
21.4.3 الكلمات على أساس موارد 436
21.4.4 دعم الاستطلاع المفتوح 439
21.5 العميل 441
21.5.1 HTML 441
21.5.2 الإجراءات 441
21.5.3 إخراج المكونات Rendering Components 443
21.5.4 الاستطلاع 445
21.5.5 التطبيق 446
21.6 تدريبات 448
21.6.1 الحفاظ على البيانات وتخزينها 448
21.6.2 إعادة ضبط حقول التعليقات 448

مقدمة

نظن حين ننشئ النظام أننا نصممه ليخدم أغراضنا ويسدّ حاجاتنا وليشبهنا، لكن الحق أنّ الحواسيب ليست مثل البشر وإنما هي إسقاط لجزء صغير جدًا منا يتعلق بالمنطق والنظام والقواعد وينشد وضوح المقصد والهدف.

— إيللِن أولمَن Ellen Ullman، في كتابها "في جوار الآلة: حب التقنية وسيئاتها" Close to the Machine: Technophilia and its Discontents.

يشرح هذا الكتاب كيفية كتابة تعليمات ووصفات يفهمها الحاسوب، ومن المعلوم أن الحواسيب شائعة ومنتشرة بحيث لا يخفى ذلك على ذي عقل وشأنها مثل شأن مفكات البراغي، غير أنها أعقد قليلًا وليس من السهل جعلها تنفذ ما تريد منها بالضبط، إلا إذا كانت المهمة التي تريد للحاسوب تنفيذها سهلة الفهم مثل عرض رسائلك البريدية أو تشغيل برنامج الحاسبة، فحينها ما عليك سوى فتح البرنامج المخصص لذلك، لكن حين تكون المهمة التي لديك فريدةً وأبعادها غير معرفة، فلن تجد تطبيقًا متاحًا لها.

يأتي هنا دور البرمجة، فهي الفعل الذي يصف بناء برنامج يتكون من مجموعة أوامر محدَّدة جدًا لتخبر الحاسوب بما يجب فعله، وتُعَدّ الحواسيب بصفتها آلات غبية لا تستطيع فهم البرمجة ولا استيعابها، لكن إذا غضضت النظر عن هذا ورأيت أنه من الممتع والمسلِّي التفكير بمنطق يشبه ذاك الذي تفهمه تلك الآلات فستجد البرمجة أمرًا مسلِّيًا جدًا ونافعًا، فهي تتيح لنا اختصار زمن الأشياء التي ننفذها يدويًا وتستغرق أوقاتًا طويلةً إلى ثواني معدودة، وعليه يمكن النظر إليها على أنها طريقة تجعل حاسوبك يقوم بأمور لم يكن يستطيعها من قبل، وهي في ذاتها -أي البرمجة- بهذا الفهم تكون تدريبًا ممتازًا على التفكير النظري المجرَّد.

كما نوجه هذه الآلات لما نريده من خلال إعطائها أوامر محدَّدة كما ذكرنا، وتكون هذه الأوامر أو البرامج من خلال لغات خاصة بالبرمجة، وهي لغات أُنشئت عمدًا لتُستخدَم في برمجة الحواسيب، ومما يعجب المرء له أن البشر في تطويرهم للأسلوب الذي يتعاملون به مع الحاسوب لم يجدوا أفضل من الطريقة التي يتواصلون بها مع

بعضهم بعضًا، فلغات البرمجة الحوسبية تشبه لغات البشر في إمكانية استخدام الكلمات والجمل في صور مختلفة لكتابة تعليمات جديدة في كل مرة تكتب برنامجًا فيها.

كانت لغة بيزيك Basic ونظام دوس DOS في الثمانينيات والتسعينيات من القرن الماضي هما الطريقتان الأساسيتان في التعامل مع الحواسيب، وهما أنظمة نصية بالكامل من غير واجهة رسومية مثل التي تراها الآن أمامك على الشاشة، وقد استبدلنا الواجهات المرئية بهما منذ ذلك الحين بما أنها أسهل في التعلم للمستخدِم رغم أنها محدودة الإمكانيات موازنة بالواجهات النصية، لكن لا زالت لغات الحاسوب موجودةً، فإذا أمعنت النظر فسترى أنّ لغةً مثل جافاسكربت JavaScript موجودة في كل متصفح تستخدِمه أنت وهي في كل حاسوب تقريبًا، وإنّ مراد هذا الكتاب الذي بين يديك هو جعلك تألف التعامل مع هذه اللغة لتستخدِمها في صنع برامج نافعة لك ولعملائك.

عن البرمجة

سنتعرض في شرحنا لهذا الكتاب للمفاهيم الأساسية للبرمجة نفسها، إذ تُعَدّ البرمجة صعبةً نوعًا ما رغم بساطة القواعد الأساسية ووضوحها، لكن البرامج التي تُبنى على تلك القواعد تصير معقدةً إلى الحد الذي يجعلها تتخذ قواعد وقوانين خاصةً بها، ويلحق ذلك بالتبعية مستوى تعقيد جديد، فنستطيع القول أنك حين تكتب برنامجًا فإنك تصنع متاهتك الخاصة التي قد تضيع فيها إذا لم تنتبه لما تكتبه.

كما نعلم أنك ستشعر بالحيرة مما قد يرد في بعض هذا الكتاب خاصةً إذا كنت جديدًا على البرمجة بما أنه سيحوي الكثير من المفاهيم التي عليك استيعابها وفهمها، وستجد أغلب تلك المفاهيم موضوعةً في تراكيب مختلفة تتطلب منك النظر فيها لفهمها، وعلى ذلك يكون عليك بذل الجهد الذي يلزمك للاستيعاب والفهم، فلا تقفز إلى استنتاج حول مستوى فهمك إذا ما أشكلت عليك فقرة أو استعصى عليك أحد المفاهيم الجديدة، وإنما لا تقلق وأرح جسدك وروِّح عن نفسك قليلًا ثم أعد قراءة ما أشكل عليك والأمثلة الموضوعة لشرحه والمفاهيم والقواعد المستخدَمة فيه، واعلم أنّ العلم لا يعطيك بعضه إلا إذا أعطيته كُلَّك، فهو عملية صعبة لكن فائدته عظيمة حين ترى ثمرته، وييسر عليك تعلُّم ما يلي منه إذا فهمته.

> اجمع معلومات وبيانات حين تكون القرارات غير مجدية؛ أما إذا كانت المعلومات غير مفيدة، فاتركها ونم إذًا.
>
> — أورسولا لي غوين Ursula K. Le Guin، رواية اليد اليسرى للظلام.

قد ترى تشبيهات كثيرة أثناء قراءتك في ماهية برنامج الحاسوب لتفهم ما هو وما وظيفته ولماذا نحتاج إليه، فبعضها يقول أنه نص كتبه مبرمِج، أو هو القوة الموجِّهة للحاسوب التي تخبره ما يجب فعله، أو البيانات المخزَّنة في ذاكرة الحاسوب، لكن رغم وجودها مخزَّنة فيها إلا أنها هي المتحكمة في الإجراءات التي تُنفَّذ على تلك الذاكرة، وهكذا فتقصر التشبيهات التي توازن البرامج بكائنات ملموسة نعرفها ونألفها عن بيان حقيقة

البرامج؛ أما التعريف السطحي للبرنامج فهو آلة تتكون من أجزاء كثيرة ومنفصلة، وعلينا نحن النظر في الطرق والأساليب التي تجعل هذه الأجزاء تعمل معًا بتناغم لتخدم غرض البرنامج الموضوع له.

يُعَدّ الحاسوب آلةً ماديةً ملموسةً تستضيف هذه الآلات غير المحسوسة -أي البرامج-، وهو في ذاته آلة غبية لا تعرف إلا تنفيذ أوامر صغيرة بسيطة وواضحة، لكن السبب الذي يجعل تلك الحواسيب نافعةً لنا هي أنها تنفِّذ تلك الأوامر بسرعة عالية جدًا، فيأتي البرنامج هنا ليستفيد من سرعة الحاسوب ويجمع له أعدادًا هائلةً من تلك الأوامر الصغيرة البسيطة ويرتبها في ترتيب منطقي يحل مشكلة لدينا نحن البشر -وما كتبناه إلا لهذا- وليجعل الحاسوب ينفذ مهامًا غاية في التعقيد، فالبرنامج إذًا هو بناء يتكون من أفكار تراكبت فوق بعضها، وهو لا يكلف موادًا وخامات لبنائه ولا وزن له ولا يشغل حيزًا كبيرًا من الفراغ بذاته رغم زيادة حجمه كلما كتبنا له أوامر جديدة لينفذها.

لكن إذا لم ننتبه للمهام التي نريدها تحديدًا من هذا البرنامج فقد يزيد حجمه وتعقيده للحد الذي يخرج عن النطاق الذي نستطيع السيطرة عليه وقد لا يستطيع حينها أن يسيطر عليه الشخص الذي كتبه نفسه، إذا يمثِّل الحفاظ على البرنامج داخل الأُطر التي توضع وفقًا للأهداف التي نكتبه من أجلها مشكلةً كبيرةً في مجال البرمجة، وعليه يمكننا القول أنّ فن البرمجة يدور حول مهارة التحكم في مدى التعقيد للبرنامج، فكلما استطعنا ترويض البرنامج ليكون أبسط فسنخرج ببرنامج يعمل بكفاءة أكثر.

يظن بعض المبرمجين أنّ حل مشكلة التعقيد تلك يكون باستخدام مجموعة محدودة من التقنيات والأساليب الواضحة التي يسهل فهمها، فكتبوا بعض القواعد الصارمة التي تصف الهيئة التي يجب أن تكون عليها البرامج وسموها الممارسات المثلى best practices مشيرين إلى قواعدهم التي وضعوها على أساس أفضل طريق لحل هذه المشكلة أو تلك، وهذا في رأينا غير فعّال ومملًا أيضًا، إذ تحتاج المشاكل الجديدة إلى حلول جديدة، كما لا يزال مجال البرمجة جديدًا نسبيًا موازنةً بالمجالات الصناعية الأخرى الضاربة في القِدم، وقد تشعب رغم ذلك كثيرًا إلى الحد الذي يسمح باتخاذ طرق مختلفة لحل المشكلة نفسها، وستجد أثناء كتابة البرامج لحلها أخطاءً فادحةً في تصميم تلك البرامج، حيث نرى أن تمضي قدمًا في طريق هذه الأخطاء لترتكبها وتقع فيها حتى تفهمها جيدًا، ذلك أنّ البرنامج الجيد برأينا قياسًا على طبيعة مجال البرمجة يُطوَّر بالمعنى الحرفي للكلمة عن طريق التجربة والخطأ وليس عن طريق اتباع قائمة من القواعد والقوانين.

سبب أهمية اللغات

كانت البرامج في بداية أمر الحواسيب تشبه ما يلي:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
```

```
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

لم يكن هناك لغات برمجة على الصورة الموجودة الآن، إذ يمثِّل الجدول أعلاه برنامجًا يجمع الأرقام من 1 إلى 10 معًا ويطبع النتيجة: `1 + 2 + ... + 10 = 55`، ولا يحتاج هذا البرنامج إلى حاسوب قوي ليعمل وإنما قد يعمل على آلة بسيطة للغاية، وقد كان من اللازم إعداد مصفوفات كبيرة من المفاتيح الحقيقية بالترتيب الصحيح من أجل كتابة برامج الحواسيب الأولى بادئ الأمر في منتصف القرن الماضي أو بطاقات مثقوبة في شرائط من الورق المقوى -أتت لاحقًا- من أجل تغذية الحاسوب بما فيها من بيانات، فتخيِّل الكفاءة الضئيلة لتلك الأنظمة المميكنة البطيئة -التي تعتمد في عملها على أجزاء ميكانيكية- والنسبة العالية للأخطاء فيها، بالإضافة إلى مدى الجهد اللازم لكتابة برنامج بسيط بمثل هذه الطرق فضلًا عن برنامج معقد.

لا شك أنّ الإدخال اليدوي لهذه الأنماط المبهمة من الأصفار والآحاد كان يجعل المبرمجين وقتها يظنون أنهم سحرة يجعلون تلك الآلات تفعل الأعاجيب بلمسات أيديهم، وإذا عدنا إلى البرنامج السابق في الجدول أعلاه، فستجد أن كل سطر يحتوي على تعليمة واحدة، وإذا كتبناها باللغة المفهومة للبشر فستكون كما يلي:

1. ضع العدد 0 في الموضع 0 من الذاكرة.
2. ضع العدد 1 في الموضع 1 من الذاكرة.
3. ضع قيمة الموضع 1 من الذاكرة داخل الموضع 2.
4. اطرح العدد 11 من القيمة الموجودة في الموضع 2 من الذاكرة.
5. انتقل إلى التعليمة رقم 9 إذا كانت قيمة الموضع 2 تساوي العدد 0.
6. أضف قيمة الموضع 1 إلى الموضع 0.
7. أضف العدد 1 إلى قيمة الموضع 1 من الذاكرة.
8. انتقل إلى التعليمة رقم 3.
9. أخرج قيمة الموضع 0.

ما زالت تعليمات البرنامج غامضةً رغم كتابتها بلغة منطوقة بدلًا من أصفار وآحاد، لهذا نستخدِم الأسماء في التعليمات البرمجية ومواضع الذاكرة بدلًا من الأعداد لتكون التعليمات أوضح وأسهل في القراءة والفهم، أي انظر كما يلي:

اضبط "الإجمالي" على 0.
اضبط "عد" على 1.
[كرر]
اضبط "وازن" على "عد".
اطرح 11 من "وازن".
إذا كانت "وازن" تساوي 0، انتقل إلى [النهاية].
أضف "عد" إلى "الإجمالي".
أضف 1 إلى "عد".
انتقل إلى [كرر].
[النهاية]
أخرج "الإجمالي".

يعطي البرنامج السابق في أول سطرين منه قيمًا أوليةً لموضعَين من الذاكرة، حيث سيستخدَم موضع "الإجمالي" ليعطينا نتيجة الحساب؛ أما "عد" فسيراقب العدد الذي ننظر إليه الآن.

أما الأسطر التي تستخدِم "وازن"، فإن البرنامج يريد أن يرى إذا كانت "عد" تساوي 11 أم لا ليقرِّر متى سيتوقف، ولا يمكنها إلا رؤية هل الرقم صفر أم لا لتتخذ إجراءً وفقًا لذلك بما أنّ آلتنا هذه بدائية نوعًا ما، وعليه فهي تستخدِم موضع الذاكرة الذي سميناه "وازن" لتحسب قيمة "عد - 11" وتتخذ إجراءً وفقًا لتلك القيمة، ثم يضيف السطران التاليان قيمة "عد" إلى النتيجة ويزيد مقدار "عد" بقيمة 1 في كل مرة يقرر البرنامج أنها لا تساوي 11 بعد. انظر الآن نفس البرنامج السابق مكتوبًا بلغة جافاسكربت، مع تبديل "total" مكان "الإجمالي"، وتبديل "count" مكان "عد":

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

تُعَدّ هذه النسخة أفضل مما سبقها من عدة أوجه لعل أهمها هو أننا لا نحتاج إلى تحديد الطريقة التي نريد البرنامج أن يقفز بها للأمام أو الخلف.

تتكفّل بُنية `while` بذلك؛ حيث تنفِّذ الكتلة المحاطة بالأقواس المعقوفة طالما تحقق الشرط المُعطى وهو أن `count <= 10` والذي يعني أن count أقل من أو يساوي 10، إذ لسنا بحاجة الآن إلى إنشاء قيمة مؤقتة وموازنتها بالصفر، وهكذا ترى أنّ من أهم مزايا لغات البرمجة أنها تتكفل بمثل هذه المهام المملة.

تُستخدَم بعد ذلك عملية console.log لكتابة النتيجة بعد إتمام بُنية while لمهمتها في نهاية البرنامج، فما رأيك الآن أن ننظر في نسخة أكفأ من أخواتها السابقات بحيث تستخدِم عملية range لتنشئ تجميعة من الأعداد داخل مجال معيَّن، ثم عملية sum التي تحسب إجمالي تلك التجميعة كما يلي:

```
console.log(sum(range(1, 10)));
// → 55
```

إن شاهد هذه الأمثلة السابقة أنّ العملية نفسها يمكن التعبير عنها بطرق طويلة أو قصيرة أو معقدة أو بسيطة، فقد كانت أول نسخة من البرنامج صعبة الفهم وطويلة، في حين أنّ آخرها كادت تكون لغةً منطوقةً بشريةً، أي: سجِّل إجمالي مجال الأعداد من 1 إلى 10.

تُعين لغة البرمجة الجيدة المبرمج على كتابة برامجه بتمكينه من الحديث عن الإجراءات التي يجب عليه تنفيذها على مستوى أعلى وإهمال التفاصيل غير الضرورية وتوفير كتل بناء فعالة مثل while وconsole.log، كما تسمح لنا بتعريف كتل بناء ننشئها نحن مثل sum وrange وتجعل استخدام مثل تلك الكتل سهلًا ومريحًا ومنطقيًا.

ما هي جافاسكربت؟

ظهرت لغة جافاسكربت لأول مرة في 1995 واستُخدِمت في إضافة البرامج إلى صفحات الويب في متصفح نِت سكيب Netscape Navigator، وقد تبنتها أغلب المتصفحات منذ ذلك الحين بما أنها جاءت بمزايا أفضل من الموجود وقتها، خاصة إمكانية التفاعل مع عناصر الصفحة الواحدة دون الحاجة إلى إعادة تحميل الصفحة في كل مرة تجري فيها تعديلًا أو تنقر فيها على زر فهي عصب الويب وتطبيقاته اليوم، واعلم أنّ لغة جافاسكربت ليس لها أيّ علاقة بلغة البرمجة التي تُسمى جافا Java، فما أتت التسمية إلا لأسباب تسويقية بحتة، إذ كانت لغة جافا ذائعة الصيت وسُوِّق لها بصورة مكثَّفة في الوقت الذي ظهرت فيه لغة جافاسكربت، ولعل الذي أطلق الاسم ظن أنها فكرة تسويقية حاذقة بجعل الاسم مشابهًا لهذه اللغة المشهورة، فكان ما حدث بعدها أن علقنا مع هذا الخلط بين اللغتين إلى الآن.

كُتِب مستند اسمه ECMAScript بعد تبني لغة جافاسكربت خارج نطاق متصفح نِت سكيب، ليكون معيارًا وتوثيقًا يصف كيف تعمل لغة جافاسكربت للبرامج التي تزعم أنها تدعمها كي تستطيع مخاطبة بعضها بعضًا باللغة نفسها، وقد سُمي بهذا الاسم بسبب منظمة Ecma الدولية التي قامت بهذه التأصيل للمعيار، وقد صار الآن مصطلح ECMAScript يشير إلى جافاسكربت والعكس بالعكس.

سترى من الناس من يقول أنّ جافاسكربت لغة سيئة بسبب كذا وكذا، وإن أغلب ما ستسمعه صحيح على الأرجح، فقد رأينا ذلك بأنفسنا عندما جربناها لأول مرة، إذ رأيناها تقبل منا أيّ شيء نكتبه ثم تفسره على عكس ما نريد، ولا شك أن هذا بسبب جهلنا في التعامل معها، لكن ذلك سببه على الحقيقة أنها مرنة جدًا في قواعدها وما تسمح به.

تكمن الفكرة التي وراء هذا التصميم في جعل البرمجة أسهل للمبتدئين، رغم أنها في الواقع تجعل البحث عن الأخطاء في برامجك أصعب لأن النظام لن يريك إياها، لكن هذه المرونة لها مزاياها أيضًا، فهي تفسح مجالًا لتقنيات وأساليب تكاد تستحيل في اللغات الأكثر صرامة كما سترى في الفصل العاشر مثلًا، إذ يمكن استخدامها للتغلب على بعض أوجه القصور في جافاسكربت، ولقد أحببنا هذه اللغة بعد العمل معها والتمرس فيها.

سنستخدِم في هذا الكتاب نسخة جافاسكربت لعام 2017، وهي نسخة حديثة ومناسبة نوعًا ما موازنة بتاريخ جافاسكربت، فحين بدأت تشتهر ويذيع صيتها كان الإصدار الثالث هو المعتمد وقتها وذلك بين عامي 2000 و2010، وقد كان العمل جاريًا وقتها على الإصدار الرابع الذي وَعد بتحديثات جديدة وتغييرات كبيرة، لكن تخلى المطورون عن أغلب تلك التغييرات في 2008، وجاء بعدها الإصدار الخامس بتحديثات أقل حجمًا في 2009، ثم خرج الذي يليه في 2015 يحمل بعضًا من المزايا التي وضِعت في الأصل للإصدار الرابع، ومنذ ذلك الوقت وجافاسكربت تحصل على تحديثات صغيرة كل عام.

يعني ذلك التطور الحاصل في اللغة أنه يجب على المتصفحات أن تتطور هي الأخرى لتواكب هذه اللغة بما أنهما مرتبطتان معًا وتدعمها، فإذا كنت تستخدِم متصفحًا قديمًا، فستجد أنّ بعض المزايا في جافاسكربت غير مدعومة رغم حرص المطورِين على عدم إجراء أيّ تغييرات تسبب أعطالًا في البرامج الموجودة الآن وذلك كي تشغِّل المتصفحات الجديدة البرامج القديمة، ولكي لا تظن أننا نضيق نطاق العمل هنا بين جافاسكربت والمتصفحات فإن المتصفحات ليست المكان الوحيد لرؤية جافاسكربت، فبعض قواعد البيانات مثل MongoDB وCouchDB تستخدِم جافاسكربت كلغة للاستعلام وكتابة السكربتات، كما تستخدِمها منصات عديدة موجهة لسطح المكتب والخوادم مثل مشروع Node.js وهو موضوع الفصل العشرين- لتوفر بيئةً لكتابة جافاسكربت واستخدامها خارج نطاق المتصفحات.

الشيفرة البرمجية واستخداماتها

الشيفرة البرمجية هي النص الذي تتكون البرامج منه، وسترى في أغلب فصول هذا الكتاب كثيرًا منها لعلمنا أنّ قراءة الشيفرات وكتابتها جزء لا يتجزأ من عملية تعلم البرمجة، كما أننا لا نريدك أن تمر عليها بعينك أثناء قراءة الكتاب وإنما قراءتها بعناية ومحاولة فهمها أيضًا، وقد يكون هذا صعبًا عليك في البداية ويجعلك تشعر بأن الأمر عسير عليك، لكننا نعدك أنك ستتعود على ذلك ليكون يسيرًا مثل قراءة هذه النصوص العادية، وبالمثل فإن ذلك ينطبق على التمارين البرمجية أيضًا فلا تفترض أنك تفهمها دون كتابة حلول لها بنفسك.

إذا قرأت الكتاب من متصفحك (بقراءة الفصول المنشورة على أكاديمية حسوب) فستستطيع تعديل كل الأمثلة البرمجية الموجودة إذا نسختها ثم نقلتها إلى بيئة اختبار برمجي مثل codepen أو codesandbox وغيرها، أو حتى في طرفية المتصفح، فرغم أنّ أمثلة كثيرة من الموجود في الكتاب ستعمل في أي بيئة

جافاسكربت، إلا أنّ الشيفرة الموجودة في الفصول الأخيرة قد كُتبت لبيئة خاصة محددة -المتصفح أو Node.js- ولن تعمل في غيرها.

هذا غير أن بعض الفصول تنظر في مشاكل متعددة وتعتمد أجزاء الشيفرات التي فيها على بعضها بعضًا أو على ملفات خارجية لبناء برامج كبيرة، وستجد في صندوق الاختبار هذا روابط إلى ملفات مضغوطة بصيغة Zip تحتوي على كل الشيفرات وملفات البيانات المطلوبة لتشغيل الشيفرة لكل فصل.

نظرة عامة على هذا الكتاب

هذا الكتاب هو النسخة العربية المترجمة عن كتاب Eloquent JavaScript الشهير لصاحبه مارين هافربيك Marijn Haverbeke، ويقع في ثلاثة أجزاء، إذ يناقش أول جزء فيها لغة جافاسكربت في اثني عشر فصلًا؛ أما الفصول السبعة التالية فهي عن متصفحات الويب والأسلوب الذي تُستخدَم لغة جافاسكربت به لبرمجتها، ثم في النهاية فصلين آخرين مخصصين لبيئة أخرى لتشغيل جافاسكربت فيها وهي Node.js، كما سيكون في هذا الكتاب خمسة فصول عملية بها مشاريع تصف برامج كبيرة لتعطيك لمحةً عن البرمجة الحقيقية، إذ سنبني روبوت توصيل ولغة برمجة ولعبة وبرنامج رسم بالبكسلات وموقعًا ديناميكيًا.

أما الجزء المتعلق باللغة نفسها فيبدأ بأربعة فصول تقدِّم البنية الأساسية للغة جافاسكربت، حيث نبدأ بشرح بنى التحكم مثل `while` التي رأيتها في المقدمة هنا، والدوال -أي كيف تكتب كتل البناء الخاصة بك- وهياكل البيانات وهنا ستتمكن من كتابة برامج بسيطة، ثم ننظر في الفصلين الخامس والسادس في تقنيات لاستخدام الدوال وكائنات لكتابة شيفرات مجردة أكثر مع ضمان السيطرة على تعقيدها.

يستمر جزء اللغة في الكتاب بعد الفصل الذي يذكر أول مشروع عملي بفصول عن معالجة الخطأ وإصلاح الزلات البرمجية Bugs، والتعابير النمطية Regular Expressions على أساس أداة مهمة للتعامل مع النصوص، والتركيبية Modularity التي تُعَدّ أحد خطوط الدفاع ضد التعقيد، والبرمجة غير المتزامنة Asynchronous programming التي تتعامل مع الأحداث التي تستغرق وقتًا، وبعد ذلك كله يلخص الفصل الخاص بثاني مشروع عملي الجزء الأول من هذا الكتاب.

ننتقل بعدها إلى الجزء الثاني مع الفصل الثالث عشر حتى التاسع عشر، حيث سنشرح فيه أدوات جافاسكربت التي يستخدِمها المتصفح، فتتعلم عرض الأشياء على الشاشة في الفصلين الرابع عشر والسابع عشر، والاستجابة لمدخلات المستخدِم في الفصل الخامس عشر، والتواصل عبر الشبكة في الفصل الثامن عشر، كما لدينا فصلين عمليين على أساس مشاريع في هذا الجزء، ثم نشرح Node.js في الفصل العشرين ونبني موقعًا صغيرًا في الفصل الذي يليه بهذه الأداة.

الاصطلاحات المطبعية في الكتاب

يمثِّل النص المكتوب بخط monospaced أي حجم ثابت للأحرف أجزاء البرامج والشيفرات البرمجية عمومًا، فإما أن تكون قطعًا مستقلةً بذاتها أو جزءًا من برنامج آخر، وتُكتب البرامج على النحو التالي:

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

لكن قد نكتب خرج البرنامج بصورة منفصلة عنه من أجل توضيحه بعد شرطتين مائلتين وسهم، أي كما يلي:

```
console.log(factorial(8));
// → 40320
```

المساهمة

يرجى إرسال بريد إلكتروني إلى academy@hsoub.com إذا كان لديك اقتراح أو تصحيح على النسخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المستعملة. إذا ضمَّنتَ جزءًا من الجملة التي يظهر الخطأ فيها على الأقل، فهذا يسهِّل علينا البحث، ,ويفضل أيضًا إضافة أرقام الصفحات والأقسام.

دورة تطوير التطبيقات
باستخدام لغة JavaScript

احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

1. القيم والأنواع والعوامل

"يسبح البرنامج تحت سطح الآلة، متمدِّدًا ومنقبِضًا من غير تكلُّف أو مشقّة، جاعلًا الإلكترونات تتناثر ثم تتجمع مرةً أخرى في تناغم عجيب، وما الأشكال والكائنات على الشاشة إلا موجات على سطح الماء، إذ يقبع أصل الحركة ومسبِّبُها متخفيًّا في الأسفل."

— يوان ما Yuan-Ma، كتاب البرمجة The Book of Programming.

لا يُنظر في عالم الحاسوب إلى شيء سوى إلى البيانات، حيث يمكنك أن تقرأ البيانات، وتُعدّلها، وتُنشِئ الجديد منها، بينما يُغفَل ذكر ما سواها، وهي متشابهة في جوهرها، إذ أنها تُخزَّن في سلاسل طويلة من البِتَّات Bits.

ويُعبَّر عن البِتَّات بأي زوج من القيم، وتُكتَب عادةً بصورة الصفر والواحد، وتأخذ داخل الحاسوب أشكالًا أخرى، مثل: شحنة كهربائية عالية أو منخفضة، أو إشارة قوية أو ضعيفة، أو ربما نقطة لامعة أو باهتة على سطح قرص مدمج CD. لذلك توصَّف أيّ معلومة فريدة، في سلسلة من الأصفار والواحدات، ومن ثم تُمثَّل في بِتَّات. فمثلًا: نمثِّل العدد 13 بالبِتَّات، بالطريقة المعتمَدة في النظام العشري، إلا أنه يوجد لكل بِتّ قيمتان فقط بدلًا من عشر قيم مختلفة، بحيث يزداد وزن كل بِتّ، ابتداءًا من اليمين إلى اليسار بمعامل العدد 2. ونحصل على البِتَّات المقابلة للعدد 13 مع بيان وزن كل بِتّ أسفل منها، كما يأتي:

1	0	1	1	0	0	0	0
1	2	4	8	16	32	64	128

وبذلك، مُثِّل العدد 13 بالعدد الثنائي 00001101، والذي نحصل عليه بجمع أوزان البِتّات الغير صفرية، أي بجمع: 8، و4، و1.

1.1 القيم

تخيّل أنّ ذاكرة الحاسوب بحر تتكوّن ذراته من بِتّات صغيرة جدًا بدلًا من ذرّات الماء، حتى أنها أشبه بمحيط كبير يتكون من 30 مليار بِتّ في ذاكرته المتطايرة volatile فقط، وأكثر من هذا بكثير في ذاكرته الدائمة (الأقراص الصلبة وما شابهها). ولكي نستطيع العمل مع هذه الأعداد الهائلة من البِتّات دون التيه في متاهة منها، فإننا نقسّمها إلى قطع كبيرة تُمثِّل أجزاءً من معلومات، وتسمّى تلك القطع في لغة جافاسكربت، بالقيم Values، ورغم أن تلك القيم تتكوّن من بِتّات، إلا أنها تختلف في وظائفها اختلافًا كبيرًا، فكل قيمة لها نوع يُحدِّد وظيفتها ودورها، فقد تكون بعض تلك القيم أعدادًا، كما قد تكون نصًّا، أو دوالًا، وهكذا. وما عليك إلا استدعاء اسم القيمة لإنشائها، دون الحاجة إلى جمع مواد لها أو شرائها، بل بمجرّد استدعاء القيمة يجعلها بين يديك.

وتُخزَّن تلك القيم في الذاكرة طبعًا، إذ يجب أن تكون في مكان ما، وقد تَنفد ذاكرة الحاسوب إذا استدعيت قيمًا كثيرة، وتحدث هذه المشكلة إذا احتجت جميع القيم التي لديك دفعةً واحدة. وتتبدّد القيم، بمجرّد انتهائك من استخدامها، تاركةً خلفها البِتّات التي شغلَتها، وذلك من أجل الأجيال المستقبلية من القيم. وسنشرح في هذا الفصل، العناصر الصغيرة جدًا في برامج جافاسكربت، وهي: أنواع القيم البسيطة، والعوامل Operators التي تعمل وفقًا لهذه القيم.

1.2 الأعداد

تُكتب القيم من النوع العددي في جافاسكربت على هيئة أعداد، أي على النحو التالي:

```
13
```

وإذا استخدمت القيمة أعلاه في برنامج ما فسيُنشِئ النمط البِتّي لها في ذاكرة الحاسوب، وتستخدم جافاسكربت، لتخزين قيمة عددية واحدة، عددًا ثابتًا من البِتّات، ومقداره تحديدًا 64 بِتّ، مما يجعل عدد الأنماط التي يمكن إنشاؤها بأربع وستين بِتّ، محدودًا نوعًا ما، وعدد الأعداد المختلفة التي يمكن تمثيلها، محدودًا أيضًا، إذ يمكنك تمثيل 10N عدد باستخدام رقم N من الأرقام العشرية، كما يمكنك تمثيل 264 عدد مختلِف باستخدام 64 رقم ثنائي أيضًا، ويُترجم هذا إلى نحو 18 كوينتليون (18 أمامها 18 صفرًا) عدد. ويُعَدّ حجم ذواكر الحاسوب قديمًا، موازنَةً بالأحجام والسعات الحالية، صغيرًا للغاية، حيث استخدم الناس، لتمثيل الأعداد، مجموعات من 8-بِتّ، أو 16-بِتّ، فكان من السهل جدًا تجاوز الحد المسموح به للأعداد التي يمكن تمثيلها فيها، لينتهي الأمر بعدد لا يمكن تمثيله ضمن العدد المعطى من البِتّات.

أما الآن، فتملك الحواسيب التي تضعها في جيبك، ذواكر كبيرةً، مما يتيح لك استخدام مجموعات من 64-بت، ولا تقلق بشأن تجاوز ذلك الحد إلا إذا تعاملت مع أعداد كبيرة جدًا. ولكن رغم ما سبق، لا يمكن تمثيل كل الأعداد الأقل من 18 كوينتليون بعدد في جافاسكربت، حيث تُخزِّن البِتّات أيضًا أعدادًا سالبة، مما يعني أنه

سيُخصَّص أحد البِتّات لتمثيل إشارة العدد، والمشكلة الأكبر هنا أنه يجب أيضًا تمثيل الأعداد الكسرية، وبالتالي سيُخصَّص بِتّ آخر لموضع الفاصلة العشرية، وذلك سيقلل عدد الأعداد التي يمكن تمثيلها بجافاسكربت، إلى 9 كوادريليون عدد (15 صفرًا هذه المرة)، وتُكتب الأعداد الكسرية fractional numbers على الصورة التالية:

```
9.81
```

وتستطيع استخدام الترميز العلمي لكتابة الأعداد الكبيرة جدًا أو الصغيرة جدًا، وذلك بإضافة e (للإشارة إلى وجود أس)، متبوعةً بأس العدد، حيث يُكتَب العدد 2.998×10^8 والمساوي لـ 299,800,000، بالشكل الآتي:

```
2.998e8
```

وتُضمَن الدقة في العمليات الحسابية المستَخدَمة مع الأعداد الصحيحة integers والأصغر من العدد المذكور آنفًا -9 كوادريليون-، على عكس العمليات الحسابية المستخدَمة مع الأعداد الكسرية، فكما أنه لا يمكن تمثيل الثابت باي π بعدد محدود من الأرقام العشرية decimal digits بعد الفاصلة، فإن كثيرًا من الأعداد تفقد دقتها عند تمثيلها بأربع وستين بِتّ فقط. ولكن لا يُمثِّل هذا مشكلةً إلا في حالات محددة. ومن المهم أن تكون على دراية بهذا، وتُعامِل الأعداد الكسرية معاملةً تقريبية، وليس على أنها قيم دقيقة.

1.2.1 الحساب

الحساب هو الشيء الأساسي المستَخدَم مع الأعداد، وتأخذ العمليات الحسابية arithmetic operations عددان، وتُنتِج عددًا جديدًا، مثل: الجمع، والضرب كما يأتي:

```
100 + 4 * 11
```

وتُسمّى الرموز مثل الرمز + والرمز * بالعوامل operators، فالعامل الأول هو عامل الجمع، والعامل الثاني هو عامل الضرب، ولدينا الرمز - للطرح، والرمز / للقسمة، وبمجرّد وضع العامل بين قيمتين، يُنفَّذ العامل الحسابي وتَنتُج قيمة جديدة، ولكن هل يعني المثال السابق أن نضيف 4 إلى 100 ونضرب النتيجة في 11، أم أن الأولوية للضرب أولًا؟ لعلّك خمّنت أن الضرب أولًا، وهذا صحيح، لكن يمكنك تغيير هذا الترتيب مثل الرياضيات، وذلك بوضع عامل الجمع بين قوسين، مما يرفع من أولوية تنفيذها، كما يأتي:

```
(100 + 4) * 11
```

ويُحدَّد ترتيب تنفيذ العوامل، عند ظهورها بدون أقواس، بأولوية تلك العوامل، فكما أن الضرب في المثال السابق له أولوية على الجمع، فللقسمة أولوية الضرب ذاتها، في حين يملك الجمع والطرح أولوية بعضهما، وتُنفَّذ العوامل بالترتيب من اليسار إلى اليمين، إذا كانت في رتبة التنفيذ نفسها، فمثلًا: إذا جاء الجمع والطرح معًا، فستكون الأولوية في التنفيذ لمن يبدأ من اليسار أولًا، أي كما في المثال التالي:

```
1 - 2 + 1
```

حيث تُنفَّذ العوامل من اليسار إلى اليمين، مثل عامل الطرح بين أقواس، أي هكذا:

```
(1 - 2) + 1
```

ولا تقلق كثيرًا بشأن هذه الأولويات، فلو حدث ونسيت أولويةً ما، أو أردت تحقيق ترتيب معيّن، فضعه داخل أقواس وانتهى الأمر.

لدينا عامل حسابي آخر قد لا تميّزه للوهلة الأولى، وهو الرمز %، والذي يُستخدَم لتمثيل عملية الباقي remainder، فيكون `Y % X` هو باقي قسمة X على Y، فمثلًا: نتيجة `314 % 100` هي 14، أما نتيجة `144 % 12` فتساوي 0. وأولوية عامل الباقي هي الأولوية ذاتها للضرب والقسمة، ويُشار عادةً إلى هذا العامل باسم modulo.

1.2.2 الأعداد الخاصة

لدينا ثلاثة قيم خاصة في جافاسكربت، ننظر إليها على أنها أعداد، ولكنها لا تُعَدّ أعدادًا طبيعية. وأول قيمتين هما: `infinity` و `-infinity` وتُمثِّلان اللانهاية بموجبها وسالبها، وبالمثل، فإن `infinity - 1` لا تزال تشير إلى اللانهاية. ولا تثق كثيرًا بالحسابات المبنيَّة على اللانهاية، لأنها ليست منطقيةً رياضيًا، وستقود إلى القيمة الخاصة التالية، وهي: `NaN`، والتي تُشير إلى "ليس عددًا" Not A Number، رغم كونه قيمةً من نوع عددي بذاته، وستحصل عليه مثلًا: إذا حاولت قسمة صفر على صفر، أو طرح لانهايتين، أو أيّ عدد من العمليات العددية التي لا تُنتِج قيمةً مفيدة.

1.3 السلاسل النصية

السلسلة النصية String هي النوع التالي من أنواع البيانات الأساسية، ويُستخدم هذا النوع لتمثيل النصوص، ويُمثَّل هذا النوع في جافاسكربت، بنص محاط بعلامات اقتباس، أي كالتالي:

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

وتستطيع استخدام العلامة الخلفية `` ` ``، أو علامات الاقتباس المفردة `'`، أو المزدوجة `''`، لتحديد السلاسل النصية، طالما أن العلامة التي وضعتها في بداية السلسلة هي ذاتها الموجودة في نهايتها. وتُوضع تقريبًا جميع أنواع البيانات داخل علامات الاقتباس تلك، وستعاملها جافاسكربت على أنها سلسلة نصية، ولكن ستجد صعوبةً في التعامل مع بعض المحارف، فمثلًا: كيف تضع علامات اقتباس داخل علامات الاقتباس المحدِّدة للسلسلة النصية؟ وكذلك محرف السطر الجديد Newlines -وهو ما تحصل عليه حين تضغط زر الإدخال Enter.

ولكتابة تلك المحارف داخل سلسلة نصية، يُنفَّذ الترميز التالي: إذا وجدت شَرطةً مائلةً خلفيةً `\` داخل نص مُقتَبس، فهذه تشير إلى أن المحرف الذي يليها، له معنى خاص، ويسمّى هذا تهريب

المحرف Escaping the character؛ إذ لن تنتهي السلسلة النصية عند احتوائها على علامات الاقتباس المسبوقة بشرطة مائلة خلفية، بل ستكون جزءًا منها؛ وحين يقع محرف n بعد شرطة مائلة خلفية فإنه يُفسَّر على أنه سطر جديد، وبالمِثل، فإذا جاء محرف t بعد شرطة مائلة خلفية، فإنه يعني محرف الجدولة tab، ومثال على ذلك، لدينا السلسلة النصية التالية:

```
"وهذا سطر جديدn\هذا سطر"
```

حيث سيبدو النص بعد تفسيره، كما يأتي:

```
هذا سطر
وهذا سطر جديد
```

كما ستحتاج في بعض المواقف إلى وضع شرطة مائلة خلفية داخل السلسلة النصية، لتكون مجرّد شرطة مائلة \، وليست مِحرفًا خاصًّا، فإذا جاءت شرطتان مائلتان خلفيتان متتابعتان، فستلغيان بعضهما، بحيث تظهر واحدة منهما فقط في القيمة الناتجة عن السلسلة النصية. فمثلًا، تُكتَب السلسلة النصية التالية: "يُكتَب محرف السطر الجديد هكذا "\n"." في جافاسكربت، كما يأتي:

```
"يُكتب محرف السطر الجديد هكذا \"\\n\"."
```

وينطبق هنا ما ذكرناه سابقًا في شأن البِتّات وتخزين الأعداد، حيث يجب تخزين السلاسل النصية على هيئة بِتّات داخل الحاسوب. تُخزِّن جافاسكربت السلاسل النصية بناءً على معيار يونيكود Unicode، الذي يُعيِّن عددًا لكل محرف تقريبًا قد تحتاج إليه ، بما في ذلك المحارف التي في اللغة العربية، واليونانية، واليابانية، والأرمنية، وغيرها.

وتُمثَّل السلسلة النصية بمجموعة من الأعداد، بما أنه لدينا عدد لكل محرف، وهذا ما تفعله جافاسكربت تحديدًا، لكن لدينا مشكلة، فتمثيل جافاسكربت يستخدِم 16بت لكل عنصر من عناصر السلسلة النصية، ما يعني أنه لدينا 216 محرفًا مختلفًا، ولكن يُعرِّف اليونيكود أكثر من ذلك، أي بمقدار الضعف تقريبًا هنا، لذا تشغل بعض المحارف مثل الرموز تعبيرية emoji، موقعين من مواقع المحارف في سلاسل جافاسكربت النصية، وسنعود لهذا مرةً أخرى في الفصل الخامس. ولا يمكن تقسيم السلسلة النصية أو ضربها أو الطرح منها، لكن يمكن استخدام عامل + عليها، حيث لا يضيف بعضها إلى بعض كما تتوقّع من +، وإنما يجمعها إلى بعضها ويسلسلها معًا، أو يلصق إن صحّ التعبير بعضها ببعض، فمثلًا، سينتج السطر التالي، كلمة "concatenate":

```
"con" + "cat" + "e" + "nate"
```

تملك القيم النصية عددًا من الدوال المصاحبة لها -التوابع methods- التي تُستخدَم لإجراء عمليات أخرى عليها، وسنذكُر هذا بمزيد من التفصيل في الفصل الرابع. حيث تتصرّف السلاسل النصية المحاطة بعلامات اقتباس مفردة أو مزدوجة تصرّفًا متشابهًا تقريبًا، والاختلاف الوحيد بينهما هو نوع الاقتباس الذي تحتاج تهريبه

داخلها. أما السلاسل المحاطة بعلامة خلفية (`)، والتي تُسمّى عادةً بالقوالب المجرّدة template literals، فيمكن تنفيذ عوامل إضافية عليها، مثل الأسطر الجديدة التي ذكرناها، أو تضمين قيم أخرى، كما في المثال التالي:

```
`half of 100 is ${100 / 2}`
```

حين تكتب شيئًا داخل `${}` في قالب مجرّد، ستُحسب نتيجته، ثم تُحوَّل هذه النتيجة إلى سلسلة نصية وتُدمَج في ذلك الموضع، وعليه يخرج المثال السابق "half of 100 is 50".

1.4 العوامل الأحادية

تُكتَب بعض العوامل على هيئة كلمات، فليست كلها رموزًا، وأحد الأمثلة على ذلك هو عامل `typeof`، والذي يُنتِج قيمةً نصيةً تُسمّي اسم نوع القيمة الممررة إليه. انظر الشيفرة التالية، تستطيع تعديلها وتشغيلها في طرفية المتصفّح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

استخدمنا `console.log` في المثال التوضيحي السابق، لبيان أننا نريد أن نرى نتيجة تقييم شيء ما، وسنبيِّن ذلك لاحقًا في الفصل التالي. وتُنفَّذ العوامل التي بيّناها في هذا الفصل اعتمادًا على قيمتين، لكن العامل `typeof` يأخذ قيمةً واحدةً فقط، وتُسمّى العوامل التي تستخدم قيمتين، بالعوامل الثنائية binary operators، أما تلك التي تأخذ عاملًا واحدًا فقط، فتسمى العوامل الأحادية unary operators، مع ملاحظة أن عامل الطرح `-` يمكن استخدامه كعامل أحادي أو ثنائي، كما في المثال التالي:

```
console.log(- (10 - 2))
// → -8
```

1.5 القيم البوليانية

حين يكون لدينا احتمالان، فمن المفيد إيجاد قيمة تفرّق بين الاحتمالين، مثل: "yes" و "no"، أو "on" و "off"، وتِستخدِم جافاسكربت النوع البوليانى Boolean لهذا الغرض، ويتكون هذا النوع من قيمتين فقط، هما: القيمة true والقيمة false، وتُكتبان بهاتين الكلمتين فقط.

1.5.1 الموازنة

انظر الطريقة التالية لإنتاج قيم بوليانية:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

علامتي > و < هما اللتان تعرفهما من الرياضيات للإشارة إلى الموازنة "أصغر من"، أو "أكبر من"، وكلاهما عاملان ثنائيّان، ويُنتِجان قيمة بوليانية تُوضِّح هل الشرط مُتحقِق أم لا، ويمكن موازنة السلاسل النصية بالطريقة نفسها، كما في المثال التالي:

```
console.log("Aardvark" < "Zoroaster")
// → true
```

تُعَدّ الطريقة التي تُرتَّب بها السلاسل النصية أبجدية في الغالب، لكن على خلاف ما قد تراه في القاموس، تكون الحروف الكبيرة أقل من الحروف الصغيرة، فمثلًا، Z أقل من a، والمحارف غير الأبجدية (!، -، ...إلخ) مدمجة أيضًا في الترتيب، وتمر جافاسكربت على المحارف من اليسار إلى اليمين موازِنةً محارف يونيكود واحدًا تلو الآخر. والعوامل الأخرى التي تُستخدَم في الموازنة هي: <= (أقل من أو يساوي)، و >= (أكبر من أو يساوي)، و == (يساوي)، و!= (لا يساوي).

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Apple" == "Orange")
// → false
```

وتوجد قيمة واحدة في جافاسكربت لا تساوي نفسها، وهي NaN ("ليس عددًا")، أي كما يأتي:

```
console.log(NaN == NaN)
// → false
```

وبما أن NaN تشير إلى نتيجة حسابية غير منطقية، فهي لن تساوي أيّ نتيجة أخرى لحساب غير منطقي.

1.5.2 العوامل المنطقية

كذلك لدينا في جافاسكربت بعض العمليات التي قد تُطبَّق على القيم البوليانية نفسها، وتدعم جافاسكربت ثلاثةً منها، وهي: and، وor، وnot، ويمكن استخدامها في منطق القيم البوليانية. ويُمثَّل عامل "and" بالرمز &&، وهو عامل ثنائي نتيجته صحيحة true إن كانت القيمتان المعطتان صحيحتان معًا.

```
console.log(true && false)
// → false
```

```
console.log(true && true)
// → true
```

أما عامل الاختيار "or"، فيُمثَّل بالرمز `||`، ويُخرِج true إذا تحققت صحة إحدى القيمتين أو كليهما، كما في المثال التالي:

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

أما "Not" فتُكتب على صورة تعجب `!`، وهي عامل أحادي يقلب القيمة المعطاة له، فالصحيح المتحقِّق منه `!true` يَخرج لنا خطأً غير متحقِّق `false`، والعكس بالعكس. وعند دمج هذه العوامل البوليانية مع العوامل الحسابية والعوامل الأخرى، فلن نستطيع تَبيُّن متى نضع الأقواس في كل حالة أو متى نحتاج إليها، والحل هنا يكون بالعلم بحال العوامل التي ذكرناها حتى الآن لشق طريقك في البرامج التي تكتبها، والشيفرات التي تقرؤها، إذ أن عامل الاختيار `||` هو أقل العوامل أولوية، ثم يليه عامل `&&`، ثم عوامل الموازنة (`>`، `==`، ...إلخ)، ثم بقية العوامل، واختيرت هذه الأسبقية أو الأولوية، كي يقل استخدام الأقواس إلى أدنى حد ممكن، انظر المثال التالي:

```
1 + 1 == 2 && 10 * 10 > 50
```

والعامل الأخير الذي لدينا ليس أحاديًّا ولا ثنائيًّا، بل i; عامل ثلاثي يعمل على ثلاث قيم، ويُكتب على صورة علامة استفهام `?`، ثم نقطتين رأسيّتين `:`، أي على الصورة التالية:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

ويُسمّى هذا العامل بالعامل الشرطي، أو العامل الثلاثي بما أنه الثلاثيُّ الوحيد في جافاسكربت، وتُحدِّد القيمة التي على يسار علامة الاستفهام، نتيجة أو خرْج هذا العامل، لتكون النتيجة هي إحدى القيمتين الآخرتين، فإذا كانت هذه القيمة `true` فالخرج هو القيمة الوسطى، وإن كانت `false` فالنتيجة هي القيمة الأخيرة التي على يمين النقطتين الرأسيّتين.

1.6 القيم الفارغة

يوجد في جافاسكربت قيمتان خاصتان تُكتبان على الصيغة null، وundefined، وتُستخدمان للإشارة إلى قيمة لا معنى لها، أو غير مفيدة، وهما قيمتان في حد ذاتهما، لكنهما لا تحملان أيّ بيانات، وستجد عمليات عدّة في هذه اللغة لا تُنتِج قيمةً ذات معنى كما سترى لاحقًا، لكنها ستُخرج القيمة undefined لأنها يجب أن تُخرِج أيّ قيمة. ولا تشغل نفسك بالاختلاف بين undefined، و null، فهما نتيجة أمر عارض أثناء تصميم جافاسكربت، ولا يهم غالبًا أيّ واحدة ستختار منهما، لذا عاملهما على أنهما قيمتان متماثلتان.

1.7 التحويل التلقائي للنوع

ذكرنا في المقدمة أن جافاسكربت تقبل أيَّ برنامج تعطيه إياها، حتى البرامج التي تُنفِّذ أمورًا غريبة، وتوضح التعبيرات التالية هذا المفهوم:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

وحين يُطبَّق عامل ما على النوع الخطأ من القيم، فستُحوِّل جافاسكربت تلك القيمة إلى النوع المطلوب باستخدام مجموعة قواعد قد لا تريدها أو تتوقعها، ويسمّى ذلك تصحيح النوع القسري type coercion. إذ تُحوِّل null في التعبير الأول من المثال السابق إلى 0، وتُحوَّل "5" إلى 5 أي من سلسلة نصية إلى عدد، أما في التعبير الثالث الذي يحوي عامل الجمع + بين نص وعدد، فنفّذت جافاسكربت الربط Concatenation قبل الإضافة العددية، وحوّلت 1 إلى "1" أي من عدد إلى نص. أما عند تحويل قيمة لا تُعبِّر بوضوح على أنها عدد إلى عدد، مثل:"five" أو undefined، فسنحصل على NaN، ولذا فإن حصلت على هذه القيمة في مثل هذا الموقف، فابحث عن تحويلات نوعية من هذا القبيل.

كذلك حين نوازن قيمتين من النوع نفسه باستخدام ==، فسيسهل توقّع الناتج، إذ يجب أن تحصل على true عند تطابق القيمتين إلا في حالة NaN، أما حين تختلف القيم، فتَستخدِم جافاسكربت مجموعة معقدّة من القواعد لتحديد الإجراء الذي يجب تنفيذه، وتحاول في أغلب الحالات أن تحوّل قيمةً أو أكثر إلى نوع القيمة

الأخرى. لكن حين تكون إحدى القيمتين `null`، أو `undefined`، فستكون النتيجة تكون صحيحةً فقط إذا كان كل من الجانبين `null` أو `undefined`. كما في المثال التالي:

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

وهذا السلوك مفيد حين تريد اختبار أيُّ القيم فيها قيمةً حقيقيةً بدلًا من `null` أو `undefined`، فتوازنهما بعامل `==` أو `!=`. لكن إن أردت اختبار شيئ يشير إلى قيمة بعينها مثل `false`، فإن التعبيرات مثل `0 == false` و `" " == false` تكون صحيحةً أيضًا، وذلك بسبب التحويل التلقائي للنوع، أما إذا كنت لا تريد حدوث أيّ تحويل نوعي، فاستخدم هذين العاملَيْن: `===`، و `!==`. حيث يَنظر أول هذين العاملَين هل القيمة مطابقة للقيمة الثانية المقابلة أم لا، والعامل الثاني ينظر أهي غير مطابقة أم لا، وعليه يكون التعبير `" " === false` خطأ كما توقعت. وإني أنصح باستخدام العامل ذي المحارف الثلاثة تلقائيًّا، وذلك لتجنُّب حدوث أي تحويل نوعي يعطّل عملك، لكن إن كنت واثقًا من الأنواع التي على جانبي العامل، فليس هناك ثمة مشكلة في استخدام العوامل الأقصر.

1.7.1 اختصار العوامل ومقاطعتها

يعالج العاملان `&&` و `||` القيم التي من أنواع مختلفة، معالجةً غريبة، إذ يحوِّلان القيمة التي على يسارهما إلى نوع بولياني لتحديد ما يجب فعله، لكنهما يعيدان إما القيمة الأصلية للجانب الأيسر أو قيمة الجانب الأيمن، وذلك وفقًا لنوع العامل، ونتيجة التحويل للقيمة اليسرى، سيُعيد عامل `||` مثلًا قيمة جانبه الأيسر إذا أمكن تحويله إلى `true`، وإلا فسيعيد قيمة جانبه الأيمن. يُحدِث هذا النتيجةَ المتوقّعة إن كانت القيم بولیانية، ونتيجةً مشابهةً إن كانت القيم من نوع آخر. كما في المثال الآتي:

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

نستطيع استخدام هذا السلوك على أنه طريقة للرجوع إلى القيمة الافتراضية، فإن كانت لديك قيمة قد تكون فارغةً، فيمكنك وضع `||` بعدها مع قيمة بدل، حيث إذا كان من الممكن تحويل القيمة الابتدائية إلى `false` فستحصل على البدل. وتنص قواعد تحويل النصوص والأعداد، إلى قيم بولیانية، على أن `0`، و`NaN`، والنص الفارغ `" "`، جميعها خطأً `false`، بينما تُعَدّ القيم الأخرى `true`، لذا فإن `-1 || 0` تخرج `-1`، و `"!?" || " "` تخرج `"!?"`.

ويتصرّف عامل && تصرّفًا قريبًا من ذلك، ولكن بطريقة أخرى، فإذا كان من الممكن تحويل القيمة اليسرى إلى false فسعيد تلك القيمة، وإلا سيعيد القيمة التي على يمينه. وهذان العاملان لهما خاصيّةً أخرى مُهمة، وهي أن الجزء الذي على يمينهما يُقيَّم عند الحاجة فقط، ففي حالة true || x ستكون النتيجة القيمة true مهما كانت قيمة x، حتى لو كانت جزءًا من برنامج يُنفِّذ شيئًا مستَهجنًا، بحيث لا تُقيَّم x عندها، ويمكن قول الشيء نفسه فيما يخص false && x والتي ستعيد القيمة false دومًا وتتجاهل x. ويسمّى هذا بالتقييم المقصور أو المختصر Short-circuit Evaluation. إذ يتصرَّف العامل الشرطي تصرّفًا قريبًا من ذلك، فالقيمة التي اختيرت في النهاية هي التي تُقيَّم فقط.

1.8 خاتمة

اطلعنا في هذا الفصل على أربعة أنواع من قيم جافاسكربت، وهي: الأرقام، والسلاسل النصية، والقيم البوليانية، والغير معرَّفة، حيث تُنشَأ هذه القيم بكتابة اسمها، كما في: true، و null، أو قيمتها، كما في: 13، و"abc"، وتُحوَّل وتُجمَع هذه القيم باستخدام عوامل أحادية، أو ثنائية، أو ثلاثية. كما رأينا عوامل حسابية مثل: + , - , * , / , % وعامل الضم النصي +، وعوامل الموازنة، وهي: == , != , === , !== , < , > , <= , >= والعوامل المنطقية، وهي: &&، و ||، إلى جانب تعرُّفنا على عدّة عوامل أحادية، مثل: - الذي يجعل العدد سالبًا، أو ! المُستخدَم في النفي المنطقي، والعامل typeof لإيجاد نوع القيمة، وكذلك العامل الثلاثي ? : الذي يختار إحدى القيمتين وفقًا لقيمة ثالثة. ويعطيك ما سبق ذكره بيانات كافيةً لتستخدم جافاسكربت على أساس حاسبة جيب صغيرة، وفي الفصل التالي، سنبدأ بربط هذه التعبيرات لنكتب برامج بسيطة بها.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

2. هيكل البرنامج

سنبدأ في هذا الفصل بفعل الأشياء التي يطلَق عليها برمجة، حيث سنتوسع في أوامر جافاسكربت لنتجاوز الأسماء وأجزاء الجمل التي رأيناها حتى الآن، وذلك لنستطيع كتابة شيفرة مفيدة ذات هدف قابل للتحقيق.

2.1 التعابير والتعليمات البرمجية

كتبنا في الفصل الأول قيمًا، وطبقنا عليها عواملًا لنحصل على قيم جديدة، حيث يُعَدّ إنشاء هذه القيم، المادة الأساسية لأي برنامج في جافاسكربت، لكن يجب وضع هذه المادة داخل هيكلٍ أكبر لتصبح مفيدة، وهذا ما سنفعله في هذا الفصل.

يُسمّى الجزء الذي يُنتِج قيمةً حقيقيةً في الشيفرة البرمجية، تعبيرًا expression، وتُعَدّ كل قيمة مكتوبة حرفيًّا، تعبيرًا، مثل: `22`، أو `psychoanalysis`، كما يُعَدّ التعبير الموجود بين قوسين أيضًا تعبيرًا، كما هو الحال بالنسبة للعامل الثنائي المُطبَّق على تعبيرين، أو العامل الأحادي المُطبَّق على تعبير واحد. وهذا يُبيّن أحد أوجه الجمال في واجهة مبنية على لغة برمجية. حيث تحتوي التعبيرات على تعبيرات أخرى، وذلك بالطريقة ذاتها المتّبعة في سرد الجمل الفرعية في لغات البشر المنطوقة، إذ تحتوي الجملة الفرعية على جملة فرعية أخرى داخلها، مما يسمح لنا ببناء تعبيرات تصف الحسابات المعقدة. فإذا استجاب تعبير لجزء من جملة، فستستجيب تعليمة من جافاسكربت لجملة كاملة. وما البرنامج إلا مجموعةٌ من التعليمات الموضوعة في قائمة مرتبة! وأبسط نوع من التعليمات، هو تعبير متبوع بفاصلة منقوطة، حيث ستحصل بذلك على برنامج بسيط، لكن لا فائدة منه في الواقع العملي، كما تستطيع تعديل الشيفرة التالية وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen:

```
1;
!false;
```

ويُستخدَم التعبير لإنتاج قيمة لتستخدمها شيفرة البرنامج فيما بعد، أما التعليمة فهي مستقِلة بذاتها، لذا تُستخدَم إذا كانت ستؤثِّر بذاتها في الشيفرة البرمجية، كأن تعرض شيئًا على الشاشة، أو تغيّر الحالة الداخلية للآلة بطريقة تُؤثّر على التعليمات التي تليها. تُدعى هذه التغييرات آثارًا جانبية Side Effects، حيث تُنتِج التعليمات الموجودة في المثال السابق، القيمتين 1 و `true`، ثم تتخلص منهما فورًا، مما لا يحدث أيّ أثر أو تغيير، وعليه فلن تلاحظ أيّ شيء إذا شغّلت هذا البرنامج. تسمح جافاسكربت لك في بعض الحالات بإهمال الفاصلة المنقوطة الموجودة في نهاية التعليمة، ومع ذلك يجب وضعها في حالات أخرى، حيث سيُعامَل السطر التالي على أنه جزء من التعليمة نفسها، لأنّ القاعدة هنا معقدة قليلًا، ومن الوارد حدوث أخطاء بسببها، وعليه فستحتاج كل تعليمة إلى فاصلة منقوطة، كما سنكتب فاصلةً منقوطةً في كل تعليمة واردة في هذا الكتاب، وننصحك أن تحذو حذونا في هذا، على الأقل إلى حين تعلمك كيفية التعامل مع غياب هذه الفاصلة.

2.2 الرابطة Binding

كيف يحافظ البرنامج على حالته الداخلية، أو يتذكر أيّ شيء؟ رأينا إلى الآن كيف ننتج قيمًا جديدة من أخرى قديمة، لكن هذا لا يغيّر القيم القديمة، ويلزمنا استخدام القيم الجديدة، وإلا فستتبدد مرةً أخرى، كما توفّر جافاسكربت شيئًا اسمه المتغير variable، أو الرابطة binding، وذلك من أجل التقاط القيم والاحتفاظ بها، كما في المثال التالي:

```
let caught = 5 * 5;
```

وهذا هو النوع الثاني من التعليمات، إذ تشير الكلمة المفتاحية `let` إلى أنّ هذه الجملة ستُحدِّد رابطة، ويتبع هذه الكلمة اسم الرابطة، كما نستطيع إعطاء قيمة لها على الفور، وذلك بإضافة عامل = مع تعبير بعده. حيث تُنشِئ التعليمة السابقة رابطةً، اسمها: `caught`، وتستخدمها لتمسك بالعدد الناتج عن ضرب 5 بـ 5، كما يُستخدَم اسم الرابطة بعد تحديدها على أساس تعبير، وقيمته هي قيمة الرابطة التي يحملها حاليًا، كما في المثال التالي:

```
let ten = 10;
console.log(ten * ten);
// → 100
```

وحين تشير رابطة ما إلى قيمة، فهذا لا يعني أنها مربوطة بتلك القيمة إلى الأبد، إذ يمكننا استخدام عامل = في أي وقت على أيّ رابطة موجودة، وذلك لفصلها عن قيمتها الحالية وربطها بقيمة جديدة. انظر إلى ما يلي:

```
let mood = "light";
console.log(mood);
// → light
mood = "dark";
```

```
console.log(mood);
// → dark
```

وإذا أردت تخيُّل الرابطات هذه، فلا تفكر فيها على أنها صناديق مثلًا، فهي أقرب لمجسّات الأخطبوط، إذ لا تحتوي على قيم وإنما تلتقطها فقط، مما يعني إمكانية إشارة رابطتين إلى القيمة ذاتها، بحيث لا يستطيع البرنامج أن يصل إلى قيم ليس لديه مرجع إليها، فإذا أردت تذكّر شيء، فستَمدّ أحد المجسّات الجديدة لإمساكه، أو ستوجّه أحد المجسات الموجودة من قبل إليه. دعنا نأخذ مثالًا آخرًا، لنقل أننا نريد تذكّر عدد الدولارات التي ما زال حسن يدين بها لك، لذلك سننشِئ رابطًا لهذا، ثم حين يسدد حسن جزءًا منها، وليكن 35$ مثلًا، فإننا سنعطي هذه الرابطة قيمةً جديدة، أي كما يأتي:

```
let HasanDebt = 140;
HasanDebt = HasanDebt - 35;
console.log(HasanDebt);
// → 105
```

لا يستطيع المجس التقاط شيء إن لم تعط قيمةً للرابطة التي عرّفتها، وإذا طلبت قيمة رابطة فارغة، فستحصل على قيمة غير معرّفة `undefined`. كذلك يمكن لتعليمة `let` واحدة تعريف عدة رابطات، لكن يجب الفصل بين الرابطات بفاصلة أجنبية `,` أي كما يلي:

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

كذلك تُستخدَم الكلمتان `var`، و `const`، لإنشاء رابطات بطريقة قريبة من `let`، أي كما في المثال التالي:

```
var name = "Ayda";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayda
```

وكلمة `var` الأولى والمختَصرة من كلمة متغير variable بالإنجليزية، هي المُصرَّح بها عن الرابطات في جافاسكربت قبل 2015، وسنعود إلى الاختلاف بينها وبين `let` في الفصل التالي، لكن نريدك أن تتذكر الآن أنها تفعل الشيء نفسه تقريبًا، رغم أننا لن نستخدمها في هذا الكتاب إلا نادرًا، وذلك بسبب خصائصها التي قد تربكك أثناء العمل. أما كلمة `const`، فهي مختصرة من كلمة ثابت الإنجليزية constant، حيث تعرِّف رابطةً ثابتةً تشير إلى القيمة ذاتها دائمًا، وهذا مستَخدم في الرابطات التي تُسنِد اسمًا إلى قيمة، وذلك لتستطيع الإشارة إليها فيما بعد بسهولة.

2.3 أسماء الرابطة

يمكن تسمية الرابطة بأي كلمة كانت، ويجوز إدخال الأرقام في أسماء الرابطات مثل `catch22`، لكن شريطة عدم بدء الاسم برقم. يمكن كذلك إدخال علامة الدولار `$` والشرطة السفلية `_` فيه، ولا يُسمح بأي علامة ترقيم، أو محرف خاص آخر. كما لا تُستَخدم الكلمات التي لها معنى خاص مثل `let`، على أساس اسم لرابطة بما أنها كلمات مفتاحية ومحجوزة، إضافةً إلى مجموعة من الكلمات المحجوزة للاستخدام في الإصدارات التالية من جافاسكربت، والقائمة الكاملة لها طويلة نوعًا ما، وهي:

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

ولا تكلِّف نفسك عناء حفظ تلك الكلمات، فإذا حدث وأنشأت رابطةً، ثم ظهر لك خطأ بناء الجملة syntax error غير متوقّع، فانظر هل كنت تعرِّف كلمةً محجوزةً أم لا.

2.4 البيئة

يُطلَق اسم البيئة environment، على مجموعة الرابطات وقيمها التي توجد في وقت ما، حيث تكون تلك البيئة غير فارغة عندما يبدأ برنامج ما، إذ تحتوي على رابطات تكون جزءًا من معيار اللغة، ورابطات توفر طُرقًا للتفاعل مع النظام المحيط باللغة، فمثلًا، إذا كنت تقرأ الآن من متصفِّح، فلديك دوال تتفاعل مع الموقع المُحمَّل حاليًّا، وتقرأ مدخلات الفأرة ولوحة المفاتيح.

2.5 الدوال

تملك العديد من القيم الموجودة في البيئة الافتراضية، نوع دالة function، والدالة هي قطعة من برنامج ملفوفة بقيمة ما، بحيث قد تُطبَّق تلك القيمة لتشغيل البرنامج الذي تلتف حوله، ففي بيئة المتصفِّح مثلًا، تحمل الرابطة المسمّاة `prompt`، دالةً تُظهِر صندوقًا حواريًا صغيرًا يطلب إدخالًا من المستخدِم، ويُستخدَم هكذا:

```
prompt("Enter passcode");
```

وناتج تنفيذ الشيفرة السابقة (سواءً في codepen أو في طرفية متصفحك)، هو مربع حوار يشبه ما تعرضه الصورة التالية:

يُسمّى تنفيذ دالة ما، استدعاءً، أو طلبًا، أو نداءً، أو تطبيقًا لها، كما تستطيع استدعاء دالة بوضع قوسين بعد تعبير ينتج قيمةَ دالة، وفي الغالب، ستجد نفسك تستخدم اسم الرابطة الذي يحمل الدالة.

تُعطى القيم التي بين أقواس إلى البرنامج داخل الدالة، وفي المثال السابق، تستخدم دالة `prompt` السلسلة النصية التي نعطيها إياها، على أساس نص تظهره في الصندوق الحِواري. وتُسمّى القيم المعطاة للدوال، وُسَطاء arguments، فقد تحتاج الدوال المختلفة، عددًا مختلفًا، أو أنواعًا مختلفةً من الوسائط. ولا تُستخدَم دالة `prompt` كثيرًا في برمجة الويب الحديثة، وذلك بسبب السيطرة على الطريقة التي سيبدو بها الصندوق الحِواري، رغم أنها مفيدة في برامج الألعاب والتجارب.

2.6 دالة console.log

استخدمنا دالة `console.log` لإخراج القيم في الأمثلة السابقة، حيث تُوفِّر أغلب أنظمة جافاسكربت بما فيها المتصفحات الحديثة كلها، وNode.js، دالة `console.log`، حيث تكتب هذه الدالة وُسطاءها لتعرضها على أداة عرض نصية، ويكون الخرج في المتصفحات، في طرفية جافاسكربت، وهو جزء مخفي في المتصفح افتراضيًّا، حيث تستطيع الوصول إليه في أغلب المتصفِّحات إذا ضغطت على F12 في ويندوز، أو command-option-I في ماك، وإن لم تستطع الوصول إليه، فابحث في قوائم المتصفِّح عن عنصر اسمه أدوات المطوِر Developer Tools، أو ما يشابهها. وسيظهر خرج `console.log` عند تشغيل أمثلة هذا الكتاب في صفحاته، أو عند تشغيل شيفرات خاصة بك، وسيظهر بعد المثال بدلًا من طرفية جافاسكربت في المتصفح.

```
let x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

ورغم عدم احتواء أسماء الرابطات على محرف النقطة، إلا أنَّ `console.log` بها واحدة، وذلك لأنها ليست اسم رابطة بسيطة، بل تعبير يجلب السجل `log` من القيمة التي تحتفظ بها اسم الرابطة `console`، وستعرف معنى ذلك بالضبط في الفصل الرابع.

2.7 القيم المعادة

يُعَدّ إظهار صناديق حوارية أو كتابة نصوص على الشاشة من الآثار الجانبية، والكثير من الدوال مفيدة وعملية بسبب تلك الآثار الجانبية التي تنتجها، كما تنتج الدوال قيمًا أيضًا، وعندها لا تحتاج أن يكون لها أثر جانبي، فهي نافعة بحد ذاتها بإنتاجها للقيم. على سبيل المثال، تأخذ دالة `Math.max` أي عدد من الوسائط العددية، وتعيد أكبرها، أي كما يأتي:

```
console.log(Math.max(2, 4));
// → 4
```

يقال للدالة التي تعيد قيمةً ما، أنها تعيد تلك القيمة، والدالة بحد ذاتها تعبير في جافاسكربت، إذ يُعَد أيّ شيء ينتج قيمةً في جافاسكربت، تعبيرًا، وهكذا يمكن استخدام استدعاءات الدوال في تعبيرات أكبر كما في المثال التالي، لاستدعاء `Math.min` الذي يعطي عكس نتيجة `Math.max`، إذ نستخدمه على أساس جزء من تعبير إضافة:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

سنشرح في الفصل التالي كيف نكتب الدوال الخاصة بنا.

2.8 تدفق التحكم

حين يحتوي برنامجك على أكثر من تعليمة واحدة، فستُنفَّذ التعليمات على أساس قصة من الأعلى إلى الأسفل، والمثال التالي فيه تعليمتان، حيث تسأل الأولى المستخدِم عن عدد، وتُنفَّذ الثانية بعدها لتُظهِر مربع ذلك العدد.

```
let theNumber = Number(prompt("اختر عددًا"));
console.log("عددك هو الجذر التربيعي لـ" +
            theNumber * theNumber);
```

حيث تحوِل دالة `Number` القيمة إلى عدد، ونحتاج هذا التحويل لأننا نريد عددًا ونتيجة `prompt` سلسلة نصية، كما توجد دوال مشابهة، اسمها: `string`، و`Boolean`، لتحويل القيم إلى تلك الأنواع أيضًا. انظر المخطط التالي الذي يوضح تدفُق التحكم لخط مستقيم:

2.9 تنفيذ شرطي

ليست كل البرامج طرقًا مستقيمة، فقد نود إنشاء طريق فرعية مثلًا، حيث يأخذ البرنامج الفرع المناسب وفقًا للموقف الذي بين يديه، ويُسمى هذا بالتنفيذ الشرطي.

ينشأ التنفيذ الشرطي بكلمة `if` المفتاحية في جافاسكربت، وفي الحالة البسيطة من هذا التنفيذ الشرطي، فإننا نريد تنفيذ شيفرة عند تحقق شرط ما، فقد نود مثلًا إظهار مربع الدخل إذا كان الدخل عددًا، أي كما يأتي:

```
let theNumber = Number(prompt("اختر عددًا"));
if (!Number.isNaN(theNumber)) {
  console.log("عددك هو الجذر التربيعي للعدد " +
              theNumber * theNumber);
}
```

وبتعديل الدخل ليكون كلمةً مثل "ببغاء"، فلن تحصل على أيّ خرج. حيث تُنفِّذ كلمة `if` المفتاحية تعليمةً ما أو تتخطاها وفقًا لقيمة التعبير البولياني، إذ يُكتب التعبير المقرِّر بعد الكلمة المفتاحية بين قوسين ويُتبع بالتعليمة المطلوب تنفيذها. وتُعَّد دالة `Number.isNaN` دالةً قياسية في جافاسكربت، حيث تُعيد `true` إذا كان الوسيط المعطى لها ليس عددًا `NaN`، كما تُعيد دالة `Number` القيمة `NaN` إذا أعطيتها نصًا لا يُمثّل عددًا صالحًا، وعليه يُترجم الشرط إلى "إن كانت القيمة `theNumber` المدخلة عددًا، افعل هذا". حيث تُغلَّف التعليمة التي تلي `if` بين قوسين معقوصين، هما: `{}`، كما في المثال السابق، ويمكن استخدام الأقواس لجمع أيّ عدد من التعليمات في تعليمة واحدة، وتُسمّى تلك المجموعة بالكتلة Block، والتي تستطيع إهمالها جميعًا في ذلك المثال بما أنها تحمل تعليمةً واحدةً فقط، لكن يستخدمهما أغلب مبرمجي جافاسكربت في كلّ تعليمة مغلفة مثل هذه، وسنتّبع هذا الأسلوب في الكتاب غالبًا، إلا في حالات شاذة عندما تكون من سطر واحد، أي كما في المثال الآتي:

```
if (1 + 1 == 2) console.log("صحيح");
// → صحيح
```

سيكون لديك في الغالب شيفرة تنتظر التنفيذ عند تحقق الشرط، وكذلك شيفرة لمعالجة الحالة الأخرى عند عدم تحققه، حيث يُمثَّل هذا المسار البديل، بالسهم الثاني في الرسم التوضيحي السابق، إذ تستطيع استخدام كلمة `else` المفتاحية مع `if` لإنشاء مساري تنفيذ منفصلين، وكل منهما بديل عن الآخر.

```
let theNumber = Number(prompt("اختر عددًا"));
if (!Number.isNaN(theNumber)) {
  console.log("عددك هو الجذر التربيعي للعدد " +
              theNumber * theNumber);
} else {
  console.log("لمَ لم تعطني عددًا؟");
}
```

أما إن كان لديك أكثر من مسارين لتختار منهما، فيمكن استخدام سلسلة أزواج من `if/else` معًا، أي كما في المثال الآتي:

```
let num = Number(prompt("اختر عددًا"));

if (num < 10) {
  console.log("صغير");
} else if (num < 100) {
  console.log("وسط");
} else {
  console.log("كبير");
}
```

سينظر البرنامج أولًا إن كان `num` أصغر من 10، فسيختار هذا الفرع، ويظهر لك `"صغير"` وانتهى الأمر؛ أما إن كان أكبر من 10، فسيأخذ مسار `else` الذي يحتوي على `if` أخرى أيضًا. فإن تحقق الشرط الثاني (`< 100`)، فهذا يعني أنّ العدد بين 10 و 100، وسيظهر لك `"وسط"`؛ أما إن لم يكن كذلك، فسيختار مسار `else` الثاني والأخير. ويوضَّح مسار هذا البرنامج بالمخطط التالي:

2.10 حلقات dog while

لنقُل أنه لدينا برنامجًا يخرج كل الأرقام الزوجية من 0 حتى 12، حيث يُكتب هذا البرنامج كما يأتي:

```
console.log(0);
console.log(2);
console.log(4);
```

```
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

لكن الفكرة من كتابة البرنامج، هو ألا نجهد أنفسنا في العمل، إذ لن يصلح هذا المنظور قطعًا إذا أردنا كل الأرقام الزوجية الأصغر من ألف. وبالتالي، سنحتاج إلى طريقة لتشغيل جزء من شيفرة عدة مرات، ويُدعى تدفُق التحكم هذا، حلقة التكرار loop.

يسمح لنا تدفق التحكم المتكرر بالعودة إلى نقطة ما في البرنامج كنا فيها من قبل، ثم تكرار حالة البرنامج الحالية، فإن جمعنا ذلك إلى اسم رابطة مفيد، وبذلك يمكننا تنفيذ ما يلي:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

حيث تنشئ الكلمة المفتاحية `while` حلقةً تكرارية، ويتبع `while` تعبيرًا داخل أقواس، ثم تعبير (البنية عمومًا تشبه الشرط `if`)، وتستمر الحلقة في التكرار طالما أنّ خرج التعبير عند تُحويله إلى النوع البولياني، هو `true`. كما توضح رابطة `number` الطريقة التي يمكن لرابطة ما تتبُّع سير البرنامج، حيث يحصل `number` على قيمة أكثر من القيمة السابقة بمقدار 2 عند كل عملية تكرار، كما يُوازَن برقم 12 في بداية الحلقة ليقرر ما إذا كان عمل البرنامج قد انتهى أم لا.

كما يمكننا كتابة برنامج يحسب قيمة 210 (مرفوعة للأس العاشر) كمثال عن برنامج يُنفِّذ أمرًا نافعًا حقًا، وسنستخدم رابطتين لكتابة هذا البرنامج، الأولى لتتبّع سير النتيجة، والثانية لحساب عدد مرات ضرب النتيجة في 2. وتختبر حلقة التكرار وصول الرابطة الثانية إلى 10، حيث تحدِّث كلا الرابطتين طالما أن الرابطة الثانية أصغر من 10.

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

ويمكن أن يبدأ العدّاد (الرابطة الثانية) من 1، ويتحقق من شرط `<= 10` لكن الأفضل أن تعتاد على بدء العد من الصفر، وذلك لأسباب مذكورة في الفصل الرابع. أما حلقة `do` فهي بنية تحكّم مماثلة لـ `while`، ولكن تختلف في نقطة واحدة فقط، حيث تُنفِّذ متنها مرةً واحدةً على الأقل، إذ تختبر شرط التوقف بعد أول تنفيذ، ويظهر ذلك الاختبار بعد متن الحلقة. كما في المثال التالي:

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

حيث يجبرك البرنامج السابق على إدخال اسم ما، وسيسألك مرةً بعد مرة إلى أن تدخل نصًّا غير فارغ، كما سيحول عامل النفي `!` القيمة إلى قيمة بوليانية قبل نفيها، وستُحوَّل جميع النصوص إلى `true` ما عدا `""`، مما يعني استمرار الحلقة بالتكرار إلى أن تُدخل اسمًا غير فارغ.

2.11 الشيفرة المزاحة

أضفنا في الأمثلة السابقة، مسافات قبل التعليمات التي تكون جزءًا من تعليمات أكبر، وذلك للإشارة إلى تبعيتها لما قبلها من تلك التعليمات، وهذه المسافات ليست ضرورية، حيث سيقبل الحاسوب البرنامج بدون هذه المسافات، كما يُعَدّ الفاصل السطري اختياريًّا، فلو كتبت برنامجًا من سطر واحد طويل جدًا، فسيقبله الحاسوب منك. أمّا دور تلك الإزاحات داخل الكتل، فهو جعل بنية الشيفرة واضحةً وبارزة، فإذا كان لدينا شيفرة كبيرة وتداخلت الكتل ببعضها البعض، فمن الصعب رؤية بداية ونهاية كل كتلة، حيث تجعل الإزاحة المناسبة الشكل المرئي للبرنامج متوافقًا مع شكل الكتل بداخله، ونفضّل استخدام مسافتين لكل كتلة مفتوحة، بينما يفضّل البعض وضع أربع مسافات، في حين يستخدم البعض الآخر زرّ tab، ولكن المهم ليس عدد المسافات، وإنما إضافة العدد ذاته لكل كتلة.

```
if (false != true) {
  console.log("That makes sense.");
  if (1 < 2) {
    console.log("No surprise there.");
  }
}
```

وستساعدك أغلب محررات الشيفرة في إزاحة السطور تلقائيًّا، وفقًا لما تكتبه، وبالمقدار المناسب.

2.12 حلقات for

تتبع أغلب الحلقات، النمط الموضح في أمثلة while، إذ تُنشأ رابطة عدّ (عدّاد) في البداية لتتبّع سير الحلقة، ثم تأتي حلقة while، وعادةً مع تعبير اختباري ليتحقق من وصول العدّاد إلى قيمته النهائية، وتُحدَّث قيمة العداد في نهاية متن الحلقة. ولأنّ هذا النمط شائع جدًا، فقد وفّرت جافاسكربت نموذجًا أقصر قليلًا، وأكثر تفصيلًا، ومتمثّلًا في حلقة for.

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
//   … etcetera
```

يطابق هذا البرنامج، المثال السابق لطباعة الأرقام الزوجية، لكن مع فرق أنّ كل التعليمات المتعلِّقة بحالة الحلقة، مجموعة معًا بعد for. حيث يجب احتواء الأقواس الموجودة بعد for على فاصلتين منقوطتين، ليُهيِّئ الجزء الذي يسبق الفاصلة المنقوطة الأولى، الحلقة من خلال تعريف رابطة لها، كما يتحقق الجزء الثاني الذي يكون تعبيرًا، من استمرارية الحلقة، ويُحدِّث الجزء الأخير حالة الحلقة بعد كل تكرار. فنجد في معظم الحالات، هذه البنية أقصر وأوضح من بنية while. وتحسب الشيفرة التالية قيمة 210، باستخدام for بدلًا عن while:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter + 1) {
  result = result * 2;
}
console.log(result);
// → 1024
```

2.13 الهروب من حلقة

ستنتهي الحلقة من تلقاء نفسها إذا كان خرج التعبير الشرطي false، ولكنه ليس الطريق الوحيد لإنهاءها، حيث توجد تعليمة خاصة لها أثر القفز نفسه إلى خارج الحلقة، وتُسمّى break، ويتوضَّح عملها من خلال البرنامج التالي، حيث يجد أول عدد أكبر أو يساوي 20، ويقبل القسمة على 7 في الوقت نفسه.

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 21
```

واستخدام عامل الباقي % هنا يعد طريقة سهلة للتحقق إن كان رقم ما يقبل القسمة على رقم آخر أم لا، فإن كان فإن باقي قسمتهما يكون صفرًا، وليس هناك جزء يتحقق من نهاية حلقة for التي في هذا المثال، هذا يعني أن حلقة for لن تقف إلا حين تُنفذ تعليمة break، فإن حذفت تعليمة break تلك أو كتبت شرط نهاية ينتج true دائمًا فسيقع برنامجك في حلقة لا نهائية ولن يقف عن العمل، وهذا لا نريده.

فإذا أنشأت حلقة لا نهائية في أحد الأمثلة السابقة أو التالية فستُسأل بعد بضع ثواني عما إن كنت تريد إيقاف الشيفرة، فإن فشل ذلك وكنت في المتصفح فيجب أن تغلق اللسان أو النافذة ثم تفتحها مرة أخرى، بل بعض المتصفحات تغلق نفسها بالكامل لتخرج من هذا الأمر.

وبالمثل فإن كلمة continue المفتاحية تشبه break في كونها تؤثر على مجرى الحلقة، فإن وجدت الحلقة كلمة continue في متنها فإن التحكم يقفز من المتن لينتقل إلى التكرار التالي للحلقة.

2.14 تحديث الرابطة بإيجاز

يحتاج البرنامج وخاصةً في الحلقات التكرارية، إلى تحديث رابطة ما، وذلك ليحفظ قيمة بناءً على القيمة السابقة لتلك الرابطة.

```
counter = counter + 1;
```

توفّر جافاسكربت اختصارًا لهذا، ويُكتب كما يأتي:

```
counter += 1;
```

وبالمثل، فهناك اختصارات لعوامل أخرى، مثل: `result *= 2` الذي يضاعف قيمة المتغير `result` أو `counter -= 1` الذي يَعُدّ تنازليًا. ويسمح هذا باختصار مثال العدّ السابق أكثر، بحيث يصبح كما يأتي:

```
for (let number = 0; number <= 12; number += 2) {
  console.log(number);
}
```

أمّا بالنسبة لـ `counter += 1`، و`counter -= 1`، فلدينا نسخ أكثر إيجازًا منهما، وهما: `counter++` و `counter--`.

2.15 الإرسال إلى قيمة باستخدام التعليمة switch

من المتوقع استخدام الشيفرة التالية:

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

تُسمى هذه البنية باسم التبديل الشرطي `switch`، وصُمّمت للتعبير عن هذا الإرسال بطريقة مباشرة -وهي مكتسبة من أسلوب جافا وC-، لكن صياغة جافاسكربت تستخدمها بغرابة، ومن الأفضل استخدامها بدلًا من سلسلة متتالية من تعليمات `if`، أي كما في المثال التالي:

```
switch (prompt("كيف حال الجو؟")) {
  case "ممطر":
    console.log("لا تنس إحضار شمسية");
    break;
  case "مشمس":
    console.log("البس ثيابًا خفيفة");
  case "غائم":
    console.log("اخرج لتتمشى");
    break;
  default:
    console.log("هذا جو غير معروف!");
    break;
}
```

حيث تستطيع وضع عدد لانهائي من الحالات case داخل الكتلة البادئة بـ switch، وسيبدأ البرنامج بالتنفيذ عند العنوان المطابق للقيمة المعطاة في switch، أو عند default في حال عدم وجود قيمة مطابقة، حيث سيكمل التنفيذ مرورًا على العناوين الأخرى إلى حين وصوله لتعليمة break. ولكن قد تجد حالةً مثل حالة "مشمس" في المثال السابق، إذ يمكن استخدامها لمشاركة جزء من الشيفرة بين الحالات -وهي اقتراح الخروج في الجو المشمس والغائم معًا-، ومن السهل نسيان كتابة تعليمة break، مما يؤدي إلى تنفيذ شيفرة لا تريدها.

2.16 الحالة الكبيرة للأحرف

لا تحتوي أسماء الرابطة على مسافات، ولكن من المفيد كتابة بضعة كلمات لوصف ما تمثله الرابطة بوضوح، والخيارات المتاحة لك في جافاسكربت هي غالبًا ما يلي:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

فالأسلوب الأول من الصعب قراءته، ونفضّل أسلوب الشرطة السفلية عنه، إذ تتّبع أغلب دوال جافاسكربت القياسية الأسلوب الأخير، وهو كتابة الحرف الأول من كل كلمة كبيرًا، عدا الحرف الأول من الكلمة الأولى، ويفعل كذلك مبرمجو اللغة ممن يستخدمونها أيضًا، وسنتّبع هذا الأسلوب في الكتاب أيضًا، إذ يجعل خلط الأساليب قراءة الشيفرة محيّرًا، لكن في حالات قليلة مثل دالة Number، فالحرف الأول من اسم الرابطة كبير، وذلك لتحديدها بأنها دالة بانية constructor، كما سنتعرّض لهذا بالتفصيل في الفصل السادس، لكن اعلم الآن أن المهم هو ألا تشغل نفسك بعدم التناغم هذا.

2.17 التعليقات

يُعَدّ التعليق جزءًا من البرنامج رغم تجاهل الحاسوب له تمامًا عند التنفيذ، إذ نستخدمه لاحتمال عدم فهم المبرمجين أو غيرهم، للرسالة أوالمَهمة التي سينفذها البرنامج عند قراءتهم للشيفرة، وقد تحتاج أنت نفسك أحيانًا إلى تسجيل بعض الملاحظات لتكون جزءًا من برنامجك، حيث يمكنك العودة إليه فيما بعد، أو لمعرفة سبب اختيارك لهذه الدالة أو تلك. وتملك جافاسكربت أسلوبين لكتابة التعليقات، إذ يمكنك كتابة تعليق من سطر واحد باستخدام اثنين من محارف الشرطة المائلة الأمامية //، ثم التعليق بعدها، أي كما في المثال التالي:

```
let accountBalance = calculateBalance(account);
// خذ من أخيك العفو واغفر ذنوبه ولا تك في كل الأمور تعاتبه
accountBalance.adjust();
// فإنك لن تلقى أخاك مهذبًا وأي امرئ ينجو من العيب صاحبه
let report = new Report();
```

```
// أخوك الذي لا ينقضُ النأيُ عهدَه ولا عند صرف الدهر يزوَرُّ جانبه
addToReport(accountBalance, report);
// وليس الذي يلقاك بالبشر والرضا وإن غبت عنه لسعتك عقاربه
```

ويستمر التعليق البادئ بالشرطتين // إلى نهاية السطر، كما يُتجاهل جزء النص الواقع بين /* و */ بكامله، بغضّ النظر عن وقوعه في سطر واحد، أو في عدة أسطر، وهذا مفيد لإضافة كتل من المعلومات المتعلقة بملف أو بجزء من البرنامج. أي كما في المثال التالي:

```
/*
 لقد وجدت هذا العدد يزحف إلى ظهر دفتري قبل مدة، ومن
 يومها وهو يظهر لي بين الحين والآخر.
 فمرةً في منتج أشتريه، ومرةً في جهات اتصالي.
 يبدو أنه صار ملازمًا لي!
*/
const myNumber = 11213;
```

2.18 خاتمة

اطلعنا على كيفية بناء البرنامج من تعليمات قد تحتوي هي نفسها على تعليمات أخرى، وقد تحتوي على تعبيرات ربما تتكون من تعبيرات أخرى.

ويعطيك وضع التعليمات بالتتالي، برنامجًا يُنفَّذ من الأعلى إلى الأسفل، كما تستطيع إدخال مغيرات لتغيير تدفق التحكم هذا باستخدام التعليمات الشرطية، مثل: if وelse وswitch؛ وتعليمات الحلقات التكرارية، مثل: while وdo وfor.

كما تستطيع استخدام الرابطة لأجزاء البيانات في الملف تحت اسم معين، وتستفيد من هذا في تتبع حالة سير برنامجك، وكذلك علمت أن البيئة هي مجموعة من الرابطات المعرَّفة، وتضع أنظمة جافاسكربت عددًا من الرابطات القياسية في بيئتك دائمًا.

تعرّفنا أيضًا على الدوال التي هي عبارة عن قيم خاصة لتغليف جزء من برنامج ما، وتُستدعى بكتابة اسمها والوسائط التي تحملها، كما يُعَدّ استدعاؤها تعبيرًا، وقد ينتج قيمة.

2.19 تدريبات

إذا لم تكن على علم بكيفية اختبار حلول هذه التدريبات فارجع إلى فصل المقدمة.

يبدأ كل تدريب بوصف مشكلة، لذا اقرأ الوصف وحاول حل التدريب، وإذا واجهتك مشكلة، فيمكنك الاطلاع على الإرشادات التي تلي التدريب، حيث ستجد الحلول الكاملة لهذه التدريبات على

https://eloquentJavaScript.net/code، وأنصحك ألا تطّلع على الحلول، إلا بعد حل التدريب، أو محاولة ذلك على الأقل بما يكفي.

2.19.1 مثلث التكرار

اكتب حلقة لاستدعاء `console.log` سبع مرات لإخراج المثلث التالي:

```
#
##
###
####
#####
######
#######
```

تستطيع إيجاد طول السلسلة النصية من خلال كتابة `.length` بعدها، أي كما في المثال التالي:

```
let abc = "abc";
console.log(abc.length);
// → 3
```

كما تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح أو بنسخها إلى codepen.

إرشادات الحل

ربما تود حل هذا المثال ببرنامج يطبع الأعداد من 1 إلى 7، انظر في المثال الذي يطبع الأعداد الزوجية أعلاه، حيث شرحنا حلقة `for` لتجري بعض التعديلات عليه ليناسب هذا المثال، بعدها انظر في التكافؤ بين الأعداد والسلاسل النصية لمحارف #، إذ تنتقل من 1 إلى 2 بإضافة 1 أي `+=1` وبالمثل فإنك ستنتقل من # إلى ## بإضافة محرف `+="#"` وعلى ذلك يكون الحل قريبًا من سلوك برنامج يطبع أرقامًا.

2.19.2 FizzBuzz

اكتب برنامجًا لاستخدام `console.log` في طباعة كل الأرقام من 1 إلى 100، مع استثناءين فقط، طباعة "Fizz" مكان الأرقام التي تقبل القسمة على 3، و"Buzz" مكان الأرقام التي تقبل القسمة على 5 وليس 3.

وحين تفعل ذلك، عدِّل برنامجك ليطبع "FizzBuzz" مكان الأرقام التي تقبل القسمة على 3 و5 معًا، مع الحفاظ على طباعة الكلمتين كما في الفقرة السابقة.

> هذا التدريب هو سؤال للمبرمجين في المقابلات الشخصية، حيث يُقصي عدد كبير منهم، لذا تزيد فرصتك في التوظيف بمجرد حلّك لهذا السؤال.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

إرشادات الحل

إذا نظرت إلى عملية المرور على الأرقام، فسترى أنها مَهمة تكرارية، بحيث يتعلق اختيار ما يجب طباعته بالتنفيذ الشرطي، وتذكر أنّ سبب استخدام عامل الباقي % هو اختبار قابلية القسمة بين عددين -أي بدون باقي-، ففي النسخة الأولى ثلاثة احتمالات لكل عدد، فسيكون عليك إنشاء سلسلة `if/else if/else`.

أما في نسخة البرنامج الثانية فلها حل بسيط وحل ذكي. فأما الحل البسيط، فهو إضافة فرع شرطي لاختبار الشرط المعطى؛ أما الحل الذكي، فهو بناء سلسلة نصية تحتوي الكلمة أو الكلمات التي يجب إخراجها، وطباعة الكلمة أو العدد إن لم يكن ثَم كلمات، مستفيدين من العامل `||`.

2.19.3 لوحة الشطرنج

اكتب برنامجًا لإنشاء سلسلة نصية، تمثِّل هذه السلسلة لوحة بأبعاد 8 ×8، وذلك باستخدام محارف الأسطر الجديدة لفصل الأسطر، بحيث يكون أول كل سطر إما مسافةً أو محرف "#"، ويجب أن تكون المحارف على هيئة لوحة شطرنج. وبتمرير السلسلة النصية إلى `console.log` يجب أن يكون الخرج مطابقًا لهذا:

```
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
```

وحين تنشئ هذا البرنامج، عليك بتعريف الرابط `size = 8` وتعديل البرنامج ليعمل مع أي حجم `size` مخرجًا شبكة من العرض والطول المعطيين.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

إرشادات الحل

تستطيع بناء سلسلة نصية بالبدء بواحدة فارغة `""` ثم إضافة محارف إليها، حيث يُكتب محرف السطر الجديد هكذا: `"\n"`، وستحتاج إلى حلقة تكرارية داخل حلقة أخرى للعمل مع بُعدين، لذا ضع أقواسًا حول متون كلا الحلقتين، وذلك لترى بداية ونهاية كل حلقة بسهولة، ولا تنس إزاحة تلك المتون إزاحةً مناسبة. ويجب أن

يتّبع ترتيب الحلقات، الترتيب الذي بنينا به السلسلة النصية، أي سطرًا سطرًا، من اليسار إلى اليمين، ومن الأعلى إلى الأسفل، فعلى ذلك تعالِج الحلقة الخارجية الأسطر. بينما تعالِج الداخلية المحارف التي في السطر، وستحتاج إلى رابطتين لتتبّع تقدمك هنا، كما تستطيع اختبار مجموع العدّاديْن زوجيًا (2 %) لمعرفة هل عليك وضع مسافة أو محرف # في أيّ موضع معطى.

ولإنهاء السطر بإضافة محرف سطر جديد، فسيكون ذلك بعد بناء السطر، أي عليك إنهاء السطر بعد الحلقة الداخلية، وداخل الحلقة الخارجية.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

3. الدوال

> يظن الناس أنّ علوم الحاسوب فن لا يحسنه إلا العباقرة، غير أنّ الحق أبعد ما يكون عن هذا، فما هم إلا جماعة من الناس يفعلون أشياءً توضع فوق بعضها وتُبنى بترتيب، كأنها أحجار صغار في حائط كبير.
>
> — دونالد نوث (Donald Knuth)

لا غنى عن الدوال في لغة جافاسكربت، إذ نستخدمها في هيكلة البرامج الكبيرة لتقليل التكرار، ولربط البرامج الفرعية بأسماء، وكذا لعزل تلك البرامج الفرعية عن بعضها، ولعل أبرز تطبيق على الدوال هو إدخال مصطلحات جديدة في اللغة. حيث يمكن إدخال أيّ مصطلح إلى لغة البرمجة من أيّ مبرمج يعمل بها، وذلك على عكس لغات البشر المنطوقة التي يصعب إدخال مصطلحات إليها، إلا بعد مراجعات واعتمادات من مجامع اللغة. وفي الواقع المشاهد، يُعَدّ إدخال المصطلحات إلى اللغة على أساس دوال، ضرورةً حتميةً لاستخدامها في البرمجة وإنشاء برامج للسوق.

فمثلًا، تحتوي اللغة الإنجليزية -وهي المكتوب بحروفها أوامر لغات البرمجة-، على نصف مليون كلمة تقريبًا، وقد لا يعلم المتحدث الأصلي لها إلا بـ 20 ألف كلمة منها فقط، وقَلَّ ما تجد لغةً من لغات البرمجة التي يصل عدد أوامرها إلى عشرين ألفًا. وعليه، ستكون المصطلحات المتوفرة فيها دقيقة المعنى للغاية، وبالتالي فهي جامدة وغير مرنة، ولهذا نحتاج إلى إدخال مصطلحات جديدة على هيئة دوال، وذلك بحسب حاجة كل مشروع أو برنامج.

3.1 تعريف الدالة

الدالة هي رابطة منتظمة، حيث تكون قيمة هذه الرابطة هي الدالة نفسها، إذ تُعرِّف الشيفرة التالية مثلًا، الثابت `square` لتشير إلى دالة تنتج مربع أي عدد مُعطَى:

```
const square = function(x) {
  return x * x;
};
console.log(square(12));
// → 144
```

وتُنشأ الدالة بتعبير يبدأ بكلمة `function` المفتاحية، كما يكون للدوال مجموعة معامِلات parameters، معامِل وحيد هو `x` حسب المثال السابق، ومتن body لاحتواء التعليمات التي يجب تنفيذها عند استدعاء الدالة، كما يُغلَّف متن الدالة بقوسين معقوصين {} حتى ولو لم يكن فيه إلا تعليمة واحدة. كذلك يجوز للدالة أن يكون لها عدة معامِلات، أو لا يكون لها أيّ معامِل، ففي المثال التالي، لا تحتوي دالة `makenoise` على أيّ معاملات، بينما تحتوي `power` على معاملين اثنين:

```
const makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const power = function(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};

console.log(power(2, 10));
// → 1024
```

وتنتج بعض الدوال قيمًا، مثل: دالتي `power`، و`square`، ولكن هذا ليس قاعدة، إذ لا تعطي بعض الدوال الأخرى قيمةً، مثل دالة `makenoise`، ونتيجتها الوحيدة هي أثر جانبي side effect. تحدِّد تعليمة `return` القيمة التي تعيدها الدالة، فحين تمر بُنية تحكُّم control -مثل التعليمات الشرطية- على تعليمة مشابهة لهذه، فستقفز مباشرةً من الدالة الحالية، وتعطي القيمة المعادة إلى الشيفرة التي استدعت الدالة. وإن لم يتبع كلمة `return` المفتاحية أيّ تعبير، فستعيد الدالة قيمة غير معرفة `undefined`، كما تعيد الدوال التي ليس فيها تعليمة `return` قيمة غير معرفة `undefined`، مثل دالة `makenoise`. تتصرف معامِلات الدالة على

أساس الرابطات المنتظمة regular bindings، غير أنّه يحدِّد مستدعي الدالة قيمتها الأولية، وليس الشيفرة التي بداخل الدالة.

3.2 الرابطات Bindings والنطاقات Scopes

نطاق الرابطة في البرنامج هو الجزء الذي تكون الرابطة ظاهرةً فيه، حيث كل رابطة لها نطاق. وإذا عرَّفنا الرابطة خارج دالة أو كتلة شيفرات، فيكون نطاق هذه الرابطة البرنامج كاملًا، ويمكنك الإشارة إلى مثل تلك الرابطات أينما تشاء، وتسمى رابطات عامة Global Bindings؛ أما الرابطات المنشأة لمعامِلات الدالة، أو المصرح عنها داخل دالة ما، فيمكن الإشارة إليها داخل تلك الدالة فقط، وعليه تسمّى رابطات محلية Local bindings، وتُنشأ نسخ جديدة من تلك الرابطات في كل مرة تُستدعَى الدالة فيها، وذلك يوفر نوعًا من العزل بين الدوال بما أنّ كل دالة تتصرف في عالمها الخاص -بيئتها المحلية-، وييسّر فهم المراد منها دون الحاجة إلى العلم بكل ما في البيئة العامة.

كما تكون الرابطات المصرح عنها باستخدام `let`، و`const` رابطات محلية لكتلة الشيفرة التي صُرح عن تلك الرابطات فيها، فإن أنشأت أحد تلك الرابطات داخل حلقة تكرارية، فلن تتمكن الشيفرات الموجودة قبل هذه الحلقة وبعدها، من رؤية تلك الرابطة. ولم يُسمح إنشاء نطاقات جديدة لغير الدوال في إصدارات جافاسكربت قبل 2015، لذا كانت الرابطات التي أُنشِئت باستخدام كلمة `var` المفتاحية، مرئيةً في كل الدالة التي تظهر فيها هذه الرابطات، أو في النطاق العام إن لم تكن داخل دالة ما. كما في المثال التالي:

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

يستطيع كل نطاق البحث في النطاق الذي يحيط به، لذا تكون `x` ظاهرة داخل كتلة الشيفرة في المثال السابق مع استثناء وجود عدة رابطات بالاسم نفسه، ففي تلك الحالة لا تستطيع الشيفرة إلا رؤية الأقرب لها، كما في المثال التالي، حيث تشير الشيفرة داخل دالة `halve` إلى `n` الخاصة بها وليس إلى `n` العامة:

```
const halve = function(n) {
  return n / 2;
```

```
};

let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

3.2.1 النطاق المتشعب

نستطيع إنشاء كتل شيفرات ودوال داخل كتل ودوال أخرى ليصبح لدينا عدة مستويات من المحلية، فمثلًا، تخرج الدالة التالية المكونات المطلوبة لصنع مقدار من الحمُّص، وتحتوي على دالة أخرى داخلها، أي كما يأتي:

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "can", "chickpeas");
  ingredient(0.25, "cup", "tahini");
  ingredient(0.25, "cup", "lemon juice");
  ingredient(1, "clove", "garlic");
  ingredient(2, "tablespoon", "olive oil");
  ingredient(0.5, "teaspoon", "cumin");
};
```

تستطيع شيفرة الدالة `ingredient` أن ترى رابطة `factor` من الدالة الخارجية، على عكس رابطتيها المحليتين الغير مرئيتين من الدالة الخارجية، وهما: `unit`، و`ingredientAmount`. ويُحدِّد مكان كتلة الشيفرة في البرنامج الرابطات التي ستكون مرئيةً داخل تلك الكتلة، حيث يستطيع النطاق المحلي رؤية جميع النطاقات المحلية التي تحتويه، كما تستطيع جميع النطاقات رؤية النطاق العام، ويُسمّى هذا المنظور لمرئية الرابطات، المراقبة المُعجَمية Lexical Scoping.

3.3 الدوال على أساس قيم

تتصرف رابطة الدالة عادةً على أساس اسم لجزء بعينه من البرنامج، وتُعرَّف هذه الرابطة مرةً واحدةً ولا تتغير بعدها، ويسهّل علينا هذا، الوقوع في الخلط بين اسم الدالة والدالة نفسها، غير أنّ الاثنين مختلفان عن بعضهما، إذ تستطيع قيمة الدالة فعل كل ما يمكن للقيم الأخرى فعله، كما تستطيع استخدامها في تعبيرات عشوائية، وتخزينها في رابطة جديدة، وتمريرها كوسيط لدالة، وهكذا. وذلك إضافةً إلى إمكانية استدعاء تلك القيمة بلا شك. وبالمثل، لا تزال الرابطة التي تحمل الدالة مجرد رابطة منتظمة regular bindung، كما يمكن تعيين قيمة جديدة لها إذا لم تكن ثابتة constant، كما في المثال الآتي:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* لا تفعل شيئًا */};
}
```

وسنناقش في الفصل الخامس بعض الأمور الشيقة التي يمكن فعلها بتمرير قيم الدالة إلى دوال أخرى.

3.4 مفهوم التصريح

توجد طريقة أقصر لإنشاء رابطة للدالة، حيث تُستخدم كلمة `function` المفتاحية في بداية التعليمة، أي كما يلي:

```
function square(x) {
  return x * x;
}
```

ويسمى هذا بتصريح الدالة function declaration، فتعرِّف التعليمة الرابطة "`square`" وتوجهها إلى الدالة المعطاة، وذلك أسهل قليلًا في الكتابة، ولا يتطلب فاصلة منقوطة بعد الدالة، لكن قد يكون هذا الأسلوب من التصريح عن الدوال خدّاعًا:

```
console.log("يقول لنا المستقبل", future());

function future() {
  return "لن تكون هناك سيارات تطير";
}
```

وعلى الرغم من أن الدالة معرَّفة أسفل الشيفرة التي تستخدمها، إلا أنها صالحة وتعمل بكفاءة، وذلك لأن تصريح الدوال ليس جزءًا من مخطط السير العادي من الأعلى إلى الأسفل، بل يتحرك إلى قمة نطاقه، ويكون متاحًا للاستخدام من قِبَل جميع الشيفرات الموجودة في ذلك النطاق، ويفيدنا هذا أمر أحيانًا لأنه يوفر حرية ترتيب الشيفرة ترتيبًا منطقيًا ومفيدًا، دون القلق بشأن الحاجة إلى تعريف كل الدوال قبل استخدامها.

3.5 الدوال السهمية Arrow Functions

لدينا مفهوم ثالث للتصريح عن الدوال، وقد يبدو مختلفًا عن البقية، حيث يستخدِم سهمًا مكتوبًا في صورة إشارة التساوي ومحرف "أكبر من"، أي على الصورة (=>)، لهذا انتبه من الخلط بينها وبين محرف "أكبر من أو يساوي"، الذي يُكتب على الصورة (>=)، ويوضح المثال التالي هذا المفهوم:

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

يأتي السهم بعد قائمة المعامِلات ويُتبع بمتن الدالة، ويكون على صورة: "هذا الدخل (المعامِلات) يُنتِج هذا الخرج (المتن)"، وحين يكون لدينا اسم معامِل واحد، فيمكنك إهمال الأقواس المحيطة بقائمة المعامِلات، وإن كان المتن تعبيرًا واحدًا بدلًا من كتلة بين قوسين، فستعيد الدالة ذلك التعبير، وعليه ينفذ التعريفين التاليين لـ square، الشيء نفسه:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

وعندما تكون الدالة السهمية بدون معامِلات على الإطلاق، فستكون قائمة معامِلاتها مجرد قوسين فارغين، أي كما في المثال التالي:

```
const horn = () => {
  console.log("Toot");
};
```

ليس ثمة سبب لوجود الدوال السهمية وتعبيرات function معًا في اللغة، إذ يملكان الوظيفة نفسها بصرف النظر عن التفاصيل الصغيرة، وسنتحدث عن ذلك في الفصل السادس، كما لم تُضَف الدوال السهمية

إلا في عـام 2015، وذلـك من أجـل السـماح بكتابـة تعبـيرات دوال صـغيرة بأسـلوب قليـل الصـياغة، حيث سنستخدمها كثيرًا في الفصل الخامس.

3.6 مكدس الاستدعاء The call stack

قد تبدو طريقة تدفُق التحكم خلال الدوال متداخلـة قليلا، انظـر المثـال التـالي للتوضـيح، حيث ينفـذ بعض الاستدعاءات من الدوال المعطاة:

```
function greet(who) {
  console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

وعند تشغيل البرنامج السابق فإن مخطط سيره يكون كالتالي:

عنـد اسـتدعاء `greet`، يقفـز التحكم إلى بدايـة تلـك الدالـة (السـطر الثـاني في الشـيفرة)، وتسـتدَعي بدورها `console.log` التي ينتقل التحكم إليها لتُنفّذ مهمتها، ثم تعود إلى المكان الذي استدعاها، وهو السـطر الرابع. ثم يستدعي السطر الذي يليه، أي `console.log` مرةً أخرى، ويصل البرنامج إلى نهايته بعد إعـادة ذلـك. وإذا أردنا تمثيل مخطط تدفق التحكم لفظيًا، فسيكون كالتالي:

```
خارج الدالة
   في greet
        في console.log
   في greet
خارج الدالة
   في console.log
خارج الدالة
```

ونظرًا لوجوب قفز الدالة إلى المكان الذي اسـتدعاها، فلابـد للحاسـوب من تـذكُّر السـياق الـذي حـدث منـه الاستدعاء، ففي إحدى الحالات أعلاه، كـان على `console.log` العـودة إلى دالـة `greet` عنـد انتهـاء تنفيـذها، بينما تعود إلى نهاية البرنامج في الحالة الأخرى. يسمى المكان الذي يخزن فيه الحاسوب هـذا السـياق، بمكـدس الاستدعاء call stack، ويُخزَّن السياق الحالي في قمة ذلـك المكـدس في كـل مـرة تُسـتدعى دالـة، كمـا تزيـل السياق الأعلى من المكدس، وتستخدمه لمتابعـة التنفيـذ عنـدما تعـود الدالـة، حيث يحتـاج هـذا المكـدس إلى مساحة في ذاكرة الحاسوب، وبما أنّ تلك المساحة محدودة، فقد يعطيـك الحاسـوب رسـالة فشـل، مثـل: عـدم وجود ذاكرة كافية في المكدس "out of stack space"، أو تكرارات تفـوق الحـد المسـموح بـه "too much recursion" حيث لدينا المثال الآتي:

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

توضح الشيفرة السابقة هذا الأمر بسؤال الحاسوب أسئلة صعبة، تجعله يتنقّل بين الدالتين ذهابًا وإيابًا إلى ما لا نهاية، وفي حالة المكدس اللانهائي هذه، فستنفذ ذاكرة الحاسوب المتاحة، أو ستطيح بالمكدس blow the stack.

3.7 الوسائط الاختيارية Optional arguments

انظر الشيفرة التالية:

```
function square(x) { return x * x; }
console.log(square(4, true, "hedgehog"));
// → 16
```

تسمح لغة جافاسكربت بتنفيذ الشيفرة السابقة دون أدنى مشكلة، ورغم أننا عرَّفنا فيها `square` بمعامِل واحد فقط، ثم استدعيناها بثلاثة معامِلات، فقد تجاهلت الوسائط الزائدة، وحسبت مربع الوسيط الأول فقط. وتستنتج من هذا أن جافاسكربت لديها سعة -إن صح التعبير- في شأن الوسائط التي تمررها إلى الدالة، فإن مررت وسائط أكثر من اللازم، فستُتجاهل الزيادة، أما إن مرّرت وسائط أقل من المطلوب، فتُسنَد المعامِلات المفقودة إلى القيمة `undefined`. وسيّئة ذلك أنك قد تُمرِّر عدد خاطئ من الوسائط، ولن تعرف بذلك، ولن يخبرك أحد ولا حتى جافاسكربت نفسها، أما حسنته فيمكن استخدام هذا السلوك للسماح لدالة أن تُستدعى مع عدد مختلف من الوسائط. انظر المثال التالي حيث تحاول دالة `minus` محاكاة معامِل `-` من خلال وسيط واحد أو وسيطين:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
```

```
// → -10
console.log(minus(10, 5));
// → 5
```

وإذا كتبنا عامل = بعد معامِل ما، ثم أتبعنا ذلك بتعبير، فستحل قيمة التعبير محل الوسيط إذا لم يكن معطى مسبقًا، إذ تجعل دالة الأس power مثلًا، وسيطها الثاني اختياريًا، فإن لم تعطها أنت ذلك الوسيط أو تمرر قيمة undefined، فسيتغير تلقائيًا إلى 2، وستتصرف الدالة مثل دالة التربيع square بالضبط كما يأتي:

```
function power(base, exponent = 2) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
}

console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

سننظر في الفصل التالي طريقة يحصل بها متن الدالة على جميع الوسائط الممررة، ويفيدنا هذا في السماح للدالة بقبول أي عدد من الوسائط، كما في console.log، إذ تُخرج كل القيم المعطاة إليها، أي:

```
console.log("C", "O", 2);
// → C O 2
```

3.8 التغليف Closure

إذا قلنا أننا نستطيع معاملة الدوال مثل قيم، وأنه يعاد إنشاء الرابطات المحلية في كل مرة تُستدعى فيها الدالة، فإننا نتساءل هنا عما يحدث لتلك الرابطات حين يصير الاستدعاء الذي أنشأها غير نشط؟

توضح الشيفرة التالية مثالًا على هذا، فهي تعرِّف دالة wrapValue، والتي تنشئ رابطةً محليةً، ثم تعيد دالةً تصل إلى تلك الرابطة وتعيدها. انظر:

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

يُعَدّ هذا الأسلوب جائزًا ومسموحًا به في جافاسكربت، ولا يزال بإمكانك الوصول إلى كلا النسختين، حيث يوضح ذلك المثال حقيقة أنّ الرابطات المحلية تُنشَأ من جديد في كل استدعاء، وأنّ الاستدعاءات المختلفة لا تدمر رابطات بعضها البعض. وتسمّى تلك الخاصية بالمغلِّف closure، أي خاصية القدرة على الإشارة إلى نسخة بعينها من رابطة محلية في نطاق محيط enclosing scope، وتسمى الدالة التي تشير إلى رابطات من نطاقات محلية حولها، بالمغلف closure. يحررك هذا السلوك من القلق بشأن دورة حياة تلك الرابطات، ويتيح لك استخدام قيم الدوال بطرق جديدة، فيمكّننا تغيير بسيط، من قلب المثال السابق إلى طريقة لإنشاء دوال تضاعِف بقيمة عشوائية، أي كما يلي:

```
function multiplier(factor) {
  return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

وبما أنّ المعامِل نفسه يُعَدّ رابطةً محليةً، فلم نعد بحاجة إلى الرابطة الصريحة `local` من دالة `wrapValue` السابقة. ويحتاج التفكير في برامج مثل هذا إلى بعض التمرس، والنموذج الذهني المعين على هذا هو التفكير في قيم الدالة على أنها تحتوي شيفرة المتن وبيئتها التي أُنشئَت فيها، وحين تُستدعى الدالة، يرى متن الدالة البيئة التي أنشئت فيها وليس البيئة التي استدعيَت فيها. وفي المثال السابق، تُستدعى الدالة `multiplier`، وتُنشئ بيئة يكون فيها المعامل `factor` مقيدًا بـ 2، وتتذكر قيمة الدالة التي تعيدها، وتكون مخزنة في `twice`، هذه البيئة، لذا حين تُستدعى فستضاعف وسيطها بمقدار 2.

3.9 التعاود Recursion

تستطيع الدالة استدعاء نفسها طالما أنها لا تكثر من ذلك إلى الحد الذي يطفح المكدس، وتسمى هذه الدالة المستدعية نفسها باسم العودية recursive ، حيث يسمح التعاود لبعض الدوال بأن تُكتب في صور مختلفة كما في المثال التالي، إذ نرى استخدامًا مختلفًا لدالة الأس power:

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

وهذا قريب من الطريقة التي يُعرَّف بها الأس عند الرياضيين، ويصف الفكرة أفضل من الصورة التكرارية looping، إذ تستدعي الدالة نفسها عدة مرات مع أسس مختلفة، لتحقيق عمليات الضرب المتكررة. ولكن ثمة مشكلة في هذا النموذج، إذ هو أبطأ بثلاث مرات من الصورة التكرارية في تطبيقات جافاسكربت، فالمرور على حلقة تكرارية أيسر، وأقل تكلفةً من استدعاء دالة عدة مرات. وهذه المسألة، أي مسألة السرعة مقابل الأناقة، لَمعضلة فريدة بين ما يناسب الإنسان وما يناسب الآلة، فلا شك أننا نستطيع زيادة سرعة البرنامج إذا زدنا حجمه وتعقيده، وإنما يقع على المبرمج تقدير كل موقف ليوازن بين هذا وذاك.

وإذا عدنا إلى حالة دالة الأس السابقة power، فأسلوب التكرار looping وهو المنظور غير الأنيق هنا، هو أبسط وأيسر في القراءة، وليس من المنطق استبدال النسخة التعاودية recursive به وإحلالها محله، لكن اعلم أنّ اختيار الأسهل، والأرخص، والأقل تكلفةً في المال والموارد الأخرى، ليس القاعدة في البرمجة، ولا يجب أن يكون كذلك، فقد يُعرض لنا موقف نتخلى فيه عن هذه الكفاءة من السهولة والسرعة في سبيل جعل البرنامج بسيطًا وواضحًا، وقد يكون فرط التفكير في الكفاءة مشتتًا لك عن المطلوب من البرنامج في الأصل، فهذا عامل آخر يعطل تصميمه.

وبحسب المرء تعقيد البرنامج ومطلوب العميل منه، فلا داعي لإضافة عناصر جديدة تزيد القلق إلى حد العجز عن التنفيذ وإتمام العمل. لهذا أنصحك بمباشرة أول كتابتك للبرنامج بكتابة شيفرة صحيحة عاملة وسهلة الفهم، وهذا رأيي في ما يجب عليك وضعه كهدف نصب عينيك، وإن أردت تسريع البرنامج فتستطيع ذلك لاحقًا بقياس أدائه ثم تحسين سرعته إن دعت الحاجة، وإن كان القلق بشأن بطء البرنامج غالبًا ليس في محله بما أن أغلب الشيفرات لا تُنفَّذ بالقدر الذي يجعلها تأخذ وقتًا ملحوظًا. فقد تجد أحيانًا مشكلات يكون حلها أسهل

باستخدام التعاود عوضًا عن التكرار، وهي المشاكل التي تتطلب استكشاف عدة فروع أو معالجتها، إذ قد تحتوي تلك الفروع بدورها على فروع أخرى، وهكذا تجد أنّ التكرار قد يكون أقل كفاءةً من التعاود أحيانًا!

يمكنك النظر إلى الأحجية التالية كمثال على هذا، فإذا بدأنا من العدد 1 وأضفنا 5 أو ضربنا في 3 باستمرار، فسينتج لدينا مجموعة لا نهائية من الأعداد. كيف تكتب دالةً نعطيها عددًا فتحاول إيجاد تسلسل عمليات الجمع والضرب التي تنتج هذا العدد؟

> يمكن التوصل إلى العدد 13 بضرب 1 في 3، ثم إضافة 5 مرتين، في حين أننا لن نستطيع الوصول إلى العدد 15 مطلقًا.

انظر الحل الآن بأسلوب التعاود:

```
function findSolution(target) {
  function find(current, history) {
    if (current == target) {
      return history;
    } else if (current > target) {
      return null;
    } else {
      return find(current + 5, `(${history} + 5)`) ||
             find(current * 3, `(${history} * 3)`);
    }
  }
  return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

لاحظ أنّ هذا البرنامج لا يزعج نفسه بالبحث عن أقصر تسلسل من العمليات، بل أي تسلسل يحقق المراد وحسب، ولأن هذا البرنامج مثال رائع على أسلوب التفكير التعاودي، فسأعيد شرحه مفصلًا إن لم تستوعب منطقه بمجرد النظر.

تنفذ دالة `find` الداخلية التعاود الحقيقي، فتأخذ وسيطين: العدد الحالي، وسلسلة نصية string لتسجل كيف وصلنا إلى هذا العدد، فإن وجدت حلًا، فستعيد سلسلةً نصيةً توضح كيفية الوصول إلى الهدف؛ أما إن لم تجد حلا بالبدء من هذا العدد، فستعيد `null`.

ولتحقيق ذلك، تنفِّذ الدالة أحد ثلاثة إجراءات:

- يُعاد العدد الحالي إن كان هو العدد الهدف، حيث يُعَد السجل الحالي طريقة للوصول إليه.
- تُعاد `null` إن كان العدد الحالي أكبر من الهدف، فليس من المنطق أن نبحث في هذا الفرع، حيث ستجعل عملية الإضافة أو الضرب، العدد أكبر مما هو عليه.
- تُعاد النتيجة إن كنا لا نزال أقل من العدد الهدف، فتحاول الدالة كلا الطريقتين اللتين تبدءين من العدد الحالي باستدعاء نفسها مرتين، واحدة للإضافة وأخرى للضرب، وتُعاد نتيجة الاستدعاء الأول إن كان أي شيء غير `null`، وإلا فيُعاد الاستدعاء الثاني بغض النظر عن إخراجها لسلسلة نصية أم `null`.

ولفهم كيفية إخراج هذه الدالة للأثر الذي نريده، دعنا ننظر في الاستدعاءات التي تُجرى على دالة `find`، عند البحث عن حل للعدد 13:

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        too big
      find(33, "(((1 + 5) + 5) * 3)")
        too big
    find(18, "((1 + 5) * 3)")
      too big
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        found!
```

لاحظ أنّ الإزاحة في المثال السابق توضح عمق مكدس الاستدعاء. حيث تستدعي `find` في أول استدعاء لها باستدعاء نفسها للبحث عن حل يبدأ بـ `1+5`، وسيتعاود هذا الاستدعاء للبحث في كل حل ينتج عددًا أقل أو يساوي العدد الهدف. وتعيد `null` إلى الاستدعاء الأول بما أنها لن تجد ما يطابق الهدف، وهنا يتدخل عامل `||` ليتسبب في الاستدعاء الذي يبحث في `1*3`، ويكون هذا البحث هو أول استدعاء تعاودي داخل استدعاء تعاودي آخَر يصيب العدد الهدف. ويعيد آخر استدعاء فرعي سلسلة نصية، وتُمرر هذه السلسلة من قبل عامليْ `||` في الاستدعاء البيني intermediate call، مما يعيد لنا الحل في النهاية.

3.10 الدوال النامية Growing Functions

لدينا في البرمجة طريقتين لإدخال الدوال في البرامج، أولاهما أن تجد نفسك تكرر كتابة شيفرة بعينها عدة مرات، وهو أمر لا شك أنك لا تريد فعله، فيزيد وجود شيفرات كثيرة من احتمال ورود أخطاء أكثر في البرنامج، ومن الإرهاق المصاحب في البحث عنها، ووقتًا أطول في قراءة الشيفرة منك ومن غيرك ممن يحاول فهم برنامجك لتعديله أو للبناء عليه، لذا عليك أخذ تلك الشيفرة المتكررة وتسميها باسم يليق بها ويعجبك، ثم تضعها في دالة.

أما الطريقة الثانية فهي حين تحتاج إلى بعض الوظائف التي لم تكتبها بعد، ويبدو أنها تستحق دالة خاصة بها، فتبدأ بتسمية هذه الدالة، ثم تشرع في كتابة متنها، وقد تبدأ في كتابة التعليمات البرمجية التي تستخدم الدالة قبل تعريف الدالةَ نفسها. واعلم أنّ مقياس وضوح المفهوم الذي تريد وضعه في هذه الدالة، هو مدى سهولة العثور على اسم مناسب للدالة! فكلما كان هدف الدالة واضحًا ومحددًا، سهل عليك تسميتها. ولنقل أنك تريد كتابة برنامج لطباعة عددين: عدد الأبقار، وعدد الدجاج في مزرعة، مع إتباع العدد بكلمة بقرة، وكلمة دجاجة بعده، ووضع أصفار قبل كلا العددين بحيث يكون طولهما دائمًا ثلاثة خانات، فهذا يتطلب دالةً من وسيطين، وهما: عدد الأبقار، وعدد الدجاج.

```
007 Cows
011 Chickens
```

وهذا يتطلب دالة من وسيطين، هما: عدد الأبقار، وعدد الدجاج.

```
function printFarmInventory(cows, chickens) {
  let cowString = String(cows);
  while (cowString.length < 3) {
    cowString = "0" + cowString;
  }
  console.log(`${cowString} Cows`);
  let chickenString = String(chickens);
  while (chickenString.length < 3) {
    chickenString = "0" + chickenString;
  }
  console.log(`${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

إذا كتبنا `.length` بعد تعبير نصي، فسنحصل على طول هذا التعبير، أو هذه السلسلة النصية، وعليه ستضيف حلقات `while` التكرارية أصفارًا قبل سلاسل الأعداد النصية، لتكون ثلاثة محارف على الأقل. وهكذا،

فقد تمت مهمتنا! ولكن لنفرض أنّ صاحبة المزرعة قد اتصلت بنا قبيل إرسال البرنامج إليها، وأخبرتنا بإضافة إسطبل إلى مزرعتها، حيث استجلبت خيولًا، وطلبت إمكانية طباعة البرنامج لبيانات الخيول أيضًا. هنا تكون الإجابة أننا نستطيع، لكن خطر لنا خاطر بينما نحن نوشك على نسخ هذه الأسطر الأربعة، ونلصقها مرةً أخرى، إذ لا بد من وجود طريقة أفضل، أي كما يأتي:

```
function printZeroPaddedWithLabel(number, label) {
  let numberString = String(number);
  while (numberString.length < 3) {
    numberString = "0" + numberString;
  }
  console.log(`${numberString} ${label}`);
}

function printFarmInventory(cows, chickens, horses) {
  printZeroPaddedWithLabel(cows, "Cows");
  printZeroPaddedWithLabel(chickens, "Chickens");
  printZeroPaddedWithLabel(horses, "Horses");
}

printFarmInventory(7, 11, 3);
```

وهنا نجحت الشيفرة، غير أنّ اسم `printZeroPaddedWithLabel` محرج نوعًا ما، إذ يجمع في وظيفة واحدة، بين كل من: الطباعة، وإضافة الأصفار، وإضافة العنوان label، لذا بدلًا من إلغاء الجزء المكرر من البرنامج. دعنا نختر مفهومًا واحدًا فقط:

```
function zeroPad(number, width) {
  let string = String(number);
  while (string.length < width) {
    string = "0" + string;
  }
  return string;
}

function printFarmInventory(cows, chickens, horses) {
  console.log(`${zeroPad(cows, 3)} Cows`);
  console.log(`${zeroPad(chickens, 3)} Chickens`);
  console.log(`${zeroPad(horses, 3)} Horses`);
```

```
}

printFarmInventory(7, 16, 3);
```

حيث تسهل الدالة ذات الاسم الجميل والواضح مثل zeroPad، على الشخص الذي يقرأ الشيفرة معرفة ما تفعله، وهي مفيدة في مواقف أكثر من هذا البرنامج خاصة، إذ تستطيع استخدامها لطباعة جداول منسقة من الأعداد.

لكن إلى أي حد يجب أن تكون الدالة التي تكتبها ذكية، بل إلى أي حد يجب أن تكون متعددة الاستخدامات؟ اعلم أنك تستطيع عمليًا كتابة أي شيء بدءًا من دالة بسيطة للغاية، حيث تحشو عددًا ليكون بطول ثلاثة محارف، إلى نظام تنسيق الأعداد المعمم والمعقد، والذي يتعامل مع الأعداد الكسرية، والأعداد السالبة، ومحاذاة الفواصل العشرية، والحشو بمحارف مختلفة، وغير ذلك.

والقاعدة هنا، هي ألا تجعل الدالة تزيد في وظيفتها عن الحاجة، إلا إذا تأكدت يقينًا من حاجتك إلى تلك الوظيفة الزائدة، فقد يكون من المغري كتابة "أُطر عمل" frameworks عامة لكل جزء من الوظائف التي تصادفها، لكنا نهيب بك ألا تستجيب لهذه الرغبة، إذ لن تنجز أيّ عمل حقيقي لأيّ عميل ولا لنفسك حتى، وإنما ستكتب شيفرات لن تستخدمها أبدًا.

3.11 الدوال والآثار الجانبية

يمكن تقسيم الدوال إلى تلك التي تُستدعَى لآثارها الجانبية side effects، وتلك التي تُستدعَى لقيمتها المعادة -رغم أنه قد يكون للدالة آثار جانبية، وقيم معادة في الوقت نفسه-، فالدالة الأولى هي دالة مساعدة في مثال المزرعة السابق، حيث تُستدعَى printZeroPaddedWithLabel لأثرها الجانبي، فتطبع سطرًا؛ أما النسخة الثانية zeroPad، فتُستدعى لقيمتها المعادة. ولا شك أنّ الحالة الثانية مفيدة أكثر من الأولى، فتكون الدوال التي تنشئ قيمًا، أسهل في إدخالها وتشكيلها في صور جديدة عن تلك التي تنتج آثارًا جانبية مباشرة.

ولدينا من ناحية أخرى، دالةً تسمى بالدالة النقية pure function، وهي نوع خاص من الدوال المنتجة للقيم، حيث لا تحتوي على آثار جانبية، كما لا تعتمد على الآثار الجانبية من شيفرة أخرى، فمثلًا، لا تقرأ هذه الدوال الرابطات العامة global bindings التي قد تتغير قيمتها. ولهذا النوع من الدوال خاصية فريدة، إذ تنتج القيمة نفسها إن استُدعيَت بالوسائط نفسها، ولا تفعل أي شيء آخر، وإضافةً إلى ما سبق، ولا يتغير معنى الشيفرة إن أزلنا استدعاء الدالة ووضعنا مكانه القيمة التي ستعيدها. وإن حدث وشككت في عمل دالة نقية، فيمكنك اختبارها ببساطة عن طريق استدعائها، واعلم أنها إذا عملت في هذا السياق، فستعمل في أي سياق آخر، إذ تحتاج الدوال غير النقية إلى دعامات أخرى لاختبارها.

لكن مع هذا، فلا داعي للاستياء عند كتابة دوال غير نقية، أو تنفيذ عمليات تطهير لحذفها من شيفراتك، فقد تكون الآثار الجانبية مفيدة، وهذا يحدث في الغالب من حالات البرمجة. فمثلًا، لا توجد طريقة لكتابة نسخة

نقية من console.log، ونحن نحتاج هذه الدالة أيما احتياج، كما سترى أثناء تمرسك في جافاسكربت لاحقًا، كذلك تسهل الآثار الجانبية من التعبير عن بعض العمليات بطريقة فعالة، لذا قد تكون الحاجة للسرعة سببًا لتجنب هذا النقاء في الدوال.

3.12 خاتمة

اطلعنا في هذا الفصل على كيفية كتابة الدوال البرمجية التي تحتاج إليها عند تنفيذ مهمة، أو وظيفة متكررة في برامجك، وذلك باستخدام كلمة function المفتاحية التي تستطيع إنشاء قيمة دالة إذا استُخدمت على أساس تعبير؛ أما إذا استُخدمت على أساس تعليمة فتكون للإعلان عن رابطة binding، وإعطائها دالة تكون قيمةً لها، كما نستطيع إنشاء الدوال أيضًا باستخدام الدوال السهمية.

```
// عرِّف f لتحمل قيمةَ دالة
const f = function(a) {
  console.log(a + 2);
};

// صرِّح عن g كدالة
function g(a, b) {
  return a * b * 3.5;
}

// قيمةُ دالةٍ أقل إسهابًا
let h = a => a % 3;
```

أحد الجوانب الرئيسية في فهم الدوال هو فهم النطاقات، حيث تنشئ كل كتلة نطاقًا جديدًا، وتكون المعامِلات والرابطات المصرَّح عنها في نطاق معين محلية وغير مرئية من الخارج. كما تتصرف الرابطات المصرَّح عنها عبر var تصرفًا مختلفًا، حيث ينتهي بهم الأمر في أقرب نطاق دالي أو في النطاق العام.

واعلم أنّ فصل المهام التي ينفذها برنامجك إلى دوال مختلفة يفيدك في انتفاء الحاجة إلى التكرار الزائد عن الحد، وسترى أنّ الدوال مفيدة في تنظيم البرنامج، إذ تجمع تجميع الشيفرة في أجزاء تنفذ أشياءً محددة.

3.13 تدريبات

3.13.1 القيمة الصغرى

تعرضنا في الفصل السابق لدالة Math.min القياسية، والتي تعيد أصغر وسيط، ويمكننا الآن بناء شيء من هذا القبيل. اكتب دالة min تأخذ وسيطين وتعيد أقل قيم لهما.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// ضع شيفرتك هنا.

console.log(min(0, 10));
// → 0
console.log(min(0, -10));
// → -10
```

إرشادات الحل

إذا واجهتك مشكلة في وضع الأقواس في مكانها الصحيح لتعرّف دالة ما، فابدأ بنسخ أحد الأمثلة التي في هذا الفصل وعدِّله.

قد تحتوي الدالة على عدة تعليمات `return`.

3.13.2 التعاود Recursion

رأينا فيما سبق عامل الباقي %، والذي يمكن استخدامه لاختبار ما إن كان العدد زوجيًا أم فرديًا باستخدام % 2، والتي تتحقق إن كان العدد يقبل القسمة على 2 أم لا. وفيما يلي طريقةً أخرى لتحديد فيما إذا كان العدد الصحيح الموجب زوجيًا أم فرديًا:

- الصفر عدد زوجي.
- الواحد عدد فردي.
- زوجية أي عدد آخر N تساوي N-2.

عرّف دالة `isEven` التعاودية المتوافقة مع الوصف، والتي يجب أن تقبل الدالة معاملًا واحدًا (عدد صحيح موجب) وتعيد قيمةً بوليانية.

اختبرها على 50 و75، وانظر كيف تتصرف في حالة 1-، ولماذا؟ وهل ثمة طريقة لإصلاح ذلك؟

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// ضع شيفرتك هنا.

console.log(isEven(50));
// → true
```

```
console.log(isEven(75));
// → false
console.log(isEven(-1));
// → ??
```

إرشادات الحل

ستكون الدالة الخاصة بك شبيهةً بدالة `find` الداخلية، في مثال `findSolution` التعاودي في هذا الفصل، مع سلسلة `if/else if/else` التي تختبر أي الحالات الثلاثة ستتحقق، حيث تنفذ `else` الأخيرة الموافقة للحالة الثالثة من الاستدعاء التعاودي، ويجب أن يحتوي كل فرع على تعليمة `return` أو يجهز قيمة ما لتُعاد.

إذا أُعطيت الدالة عددًا سالبًا فستستدعي نفسها مرة بعد مرة ممررة أعدادًا سالبة أكثر إلى نفسها، فتزيد في بعدها عن النتيجة المعادة، وستنفد مساحة التكديس في النهاية وتنهي العملية.

3.13.3 عد حبات الفول

تستطيع الحصول على المحرف رقم N من سلسلة نصية بكتابة `string"[N]"` وستكون القيمة المُعادة سلسلة تحتوي على محرف واحد فقط، `"b"` مثلًا. سيكون للمحرف الأول الموضع 0، كما سيكون سببًا في العثور على آخر محرف في الموضع `string.length - 1`. أي ستكون محارف السلسلة المكونة من محرفين ذات طول مقداره 2، في الموضعين 0، و1.

اكتب دالة `CountBs` التي تأخذ وسيطًا وحيدًا لها وهو السلسلة النصية، وتُعيد عددًا يشير إلى عدد الأحرف الكبيرة "B" الموجودة في السلسلة، ثم اكتب دالة `countchar`، بحيث تتصرف مثل `countBs`، إلا أنها تأخذ وسيطًا ثانيًا يشير إلى المحرف المراد عدّه (بدلاً من حساب عدد الأحرف "B" الكبيرة فقط).

أعد كتابة `countBs` للاستفادة من هذه الدالة الجديدة.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح أو بنسخها إلى codepen.

```
// ضع شيفرتك هنا.

console.log(countBs("BBC"));
// → 2
console.log(countChar("kakkerlak", "k"));
// → 4
```

إرشادات الحل

ستحتاج دالتك إلى حلقة تكرارية تنظر في كل محرف داخل السلسلة النصية، ويمكن أن تشغّل فهرسًا index من الصفر إلى أقل من طول السلسلة بمقدار 1 (`< string.length`)، فإذا كان المحرف الذي في الموضع الحالي هو المحرف نفسه الذي تبحث عنه الدالة فإنه يضيف 1 إلى متغير العدّاد، ويعاد العدّاد بمجرد انتهاء الحلقة التكرارية.

تأكد من جعل كل الرابطات المستخدَمة في الدالة محلية لها، وذلك عبر تصريحها باستخدام `let` أو `cons`.

مستقل
mostaql.com

ادخل سوق العمل ونفذ المشاريع باحترافية
عبر أكبر منصة عمل حر بالعالم العربي

ابدأ الآن كمستقل

4. هياكل البيانات: الكائنات والمصفوفات

لقد سئلت مرتين من قبل أني لو أَدخلتُ إلى الآلة أرقامًا خاطئةً فهل سأحصل على إجابات صحيحة؟
وإني في الحقيقة لعاجز عن فهم هذا الخلط في الفِكر الذي يجعل العقل يثير مثل هذا السؤال.

— تشارلز بابج (Charles Babbage)، مقاطع من حياة فيلسوف (1864).

تشكل الأعداد، والقيم البوليانية، والسلاسل، الذرات التي تُبنى منها هياكل البيانات في مجال البرمجة وعلوم الحاسوب، وستحتاج عند عملك في البرمجة إلى أكثر من ذرة واحدة، إذ تسمح لنا الكائنات objects بتجميع القيم -بما في ذلك الكائنات الأخرى-، من أجل بناء هياكل أكثر تعقيدًا.

كانت البرامج التي أنشأناها حتى الآن في فصول هذا الكتاب محدودة إذ عملت فقط على أنواع بسيطة من البيانات، وسنتحدّث في هذا الفصل عن الهياكل الأساسية للبيانات، كما ستعرف بنهايته ما يكفيك للبدء بكتابة برامج مفيدة، وسنمر فيه على بعض الأمثلة الواقعية نوعًا ما، متعرضين للمفاهيم أثناء تطبيقها على المشاكل المطروحة، كما سنبني الشيفرة التوضيحية غالبًا على الدوال والروابط التي شُرحت من قبل.

4.1 الإنسان المتحول إلى سنجاب

يتوهم سمير أنه يتحوّل إلى قارض فروي صغير ذو ذيل كثيف، وذلك من حين لآخر، وغالبًا بين الساعة الثامنة والعاشرة مساءً. وهو سعيد نوعًا ما لأنه غير مصاب بالنوع الشائع للاكتيريا السريرية classic lycanthropy، أو الاستذئاب -وهي حالة تجعل الشخص يتوهم أنه يتحول إلى حيوان، ويكون ذئبًا في الغالب-، فالتحول إلى سنجاب أهون من التحول إلى ذئب! إذ ليس عليه القلق إلا من أن يؤكل من قطة جاره، بدلاً من خشية أكل جاره بالخطأ.

لقد قرر إغلاق أبواب غرفته ونوافذها في الليل، ووضع بعض حبات البندق على الأرض، وذلك بعد مرتين من إيجاد نفسه يستيقظ على غصن رقيق غير مستقر في شجرة بلوط، عاري الجسد مشوش الذهن.

ولعلّ ذلك تكفّل بمسألتي القطة وشجرة البلوط، غير أنّ سمير يريد معالجة نفسه من حالته هذه بالكليّة، وقد لاحظ أنّ حالات التحول تلك غير منتظمة، فلا بد من وجود شيء ما يستفزها، وقد ظن لفترة أنّ شجرة البلوط هي السبب، إذ حدثت بضع مرات بجانبها، لكن تبين له أنّ تجنب أشجار البلوط لم يوقف المشكلة.

رأى سمير أن يغّير منهجه في التفكير إلى سلوك علمي، فبدأ بتسجيل يومي لكل ما يفعله في اليوم وما إن كان قد تحوّل أم لا، وهو يأمل بهذه البيانات حصر الظروف التي تؤدي إلى التحولات. والآن، بإسقاط ما سبق على موضوع هذا الفصل، فأول شيء يحتاج إليه هو بنية بيانات لتخزين هذه المعلومات، أليس كذلك؟

4.2 مجموعات البيانات

إذا أردت العمل مع كميات كبيرة من البيانات الرقمية، فعليك أولًا إيجاد طريقة لتمثيلها في ذاكرة الحواسيب، فعلى سبيل المثال، إذا أردنا تمثيل تجميعة من الأرقام 2، و3، و5، و7، و11، فسنستطيع حل الأمر بأسلوب مبتكر باستخدام السلاسل النصية -إذ لا حد لطول السلسلة النصية، وعليه نستطيع وضع بيانات كثيرة فيها- واعتماد `"2 3 5 7 11"` على أنه التمثيل الخاص بنا، لكن هذا منظور غريب ومستهجن، إذ يجب استخراج الأعداد بطريقة ما، وإعادة تحويلها إلى أعداد من أجل الوصول إليها.

توفر جافاسكربت بدلًا من السلوك السابق، نوع بيانات يختص بتخزين سلاسل القيم، وهو المصفوفة array، والتي تُكتب على أساس قائمة من القيم بين قوسين مربعين، ومفصولة بفاصلات إنجليزية Comma، انظر كما يلي:

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

كذلك تَستخدِم الصيغة التي نحصل بها على العناصر داخل مصفوفة ما، الأقواس المربعة أيضًا، حيث يأتي قوسان مربعان بعد تعبير ما مباشرةً، ويحملان بينهما تعبيرًا آخرًا، إذ يبحثان في التعبير الأيسر عن عنصر يتوافق مع الفهرس index المعطى في التعبير المحصور بينهما.

الفهرس الأول للمصفوفة هو الصفر وليس الواحد، لذا يُسترد العنصر الأول باستخدام `listOfNumbers[0]`، وإن كنت جديدًا على علوم الحاسوب، فسيمر وقت قبل اعتياد بدء العد على أساس

الصفر، وهو تقليد قديم في التقنية وله منطق مبني عليه، لكن كما قلنا، ستأخذ وقتك لتتعود عليه؛ ولتريح نفسك، فكر في الفهرس على أنه عدد العناصر التي يجب تخطيها بدءًا من أول المصفوفة.

4.3 الخصائص

رأينا في الفصول السابقة بعض التعبيرات المثيرة للقلق، مثل: `myString.length` التي نحصل بها على طول السلسلة النصية، و `Math.max` التي تشير إلى الدالة العظمى، حيث تصل هذه تعبيرات إلى خصائص قيمة ما. ففي الحالة الأولى نصل إلى خاصية الطول للقيمة الموجودة في `myString`، أما في الثانية فسنصل إلى الدالة العظمى `max` في كائن `Math`، وهو مجموعة من الثوابت والدوال الرياضية.

تحتوي جميع قيم جافاسكربت تقريبًا على خصائص، باستثناء `null`، و`undefined`، إذ ستحصل على خطأ إذا حاولت الوصول إلى خاصية إحدى هذه القيم.

```
null.length;
// → TypeError: null has no properties
```

الطريقتان الرئيسيتان للوصول إلى الخصائص في جافاسكربت، هما: النقطة `.`، والأقواس المربعة `[]` حيث تصل كل من `value.x`، و`value[x]` مثلًا، إلى خاصية ما في `value`، لكن ليس إلى الخاصية نفسها بالضرورة، ويكمن الفرق في كيفية تفسير `x`، فحين نستخدم النقطة فإن الكلمة التي تليها هي الاسم الحرفي للخاصية؛ أما عند استخدام الأقواس المربعة فيُقيَّم التعبير الذي بين الأقواس للحصول على اسم الخاصية.

في حين تجلب `value.x` خاصيةً اسمها `x` لـ `value`، فستحاول `value[x]` تقييم التعبير `x`، وتستخدم النتيجة -المحوَّلة إلى سلسلة نصية- على أساس اسم للخاصية، لذا فإن كنت على علم بأنّ الخاصية التي تريدها تحمل الاسم "color"، فتقول `value.color`؛ أما إن أردت استخراج الخاصية المسماة بالقيمة المحفوظة في الرابطة `i`، فتقول `value[i]`.

واعلم أنّ أسماء الخصائص ما هي إلا سلاسل نصية، فقد تكون أي سلسلة نصية، لكن صيغة النقطة لا تعمل إلا مع الأسماء التي تبدو مثل أسماء رابطات صالحة. فإذا أردت الوصول إلى خاصية اسمها "2" أو "John Doh"، فيجب عليك استخدام الأقواس المربعة: `value[2]`، أو `value["John Doh"]`.

تُخزَّن العناصر في المصفوفة على أساس خصائص لها، باستخدام الأعداد على أساس أسماء للخصائص، وبما أنك لا تستطيع استخدام الصياغة النقطية مع الأرقام، وتريد استخدام رابطة تحمل الفهرس، فيجب عليك استخدام صيغة الأقواس للوصول إليها.

تخبرنا خاصية `length` للمصفوفة كم عدد العناصر التي تحتوي عليها، واسم الخاصية ذاك هو اسم رابطة صالح، كما نعرّف اسمه مسبقًا، لذلك نكتب `array.length` للعثور على طول المصفوفة، وذلك أسهل من كتابة `array["length"]`.

4.4 التوابع Methods

تحتوي قيم السلاسل النصية وقيم المصفوفات على عدد من الخصائص التي تحمل قيمًا للدالة، إضافةً إلى خاصية length كما في المثال التالي:

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

كل سلسلة لها خاصية toUpperCase، إذ تعيد عند استدعائها نسخةً من السلسلة التي تم فيها تحويل جميع الأحرف إلى أحرف كبيرة. وبالمثل، تسير خاصية toLowerCase في الاتجاه العكسي. ومن المثير أنّ الدالة لديها وصول لسلسلة "Doh" النصية، وهي القيمة التي استدعينا خاصيتها، رغم أن استدعاء toUpperCase لا يمرر أي وسائط، وسننظر في تفصيل كيفية حدوث ذلك في الفصل السادس.

تسمى الخصائص التي تحتوي على دوال توابعًا للقيم المنتمية إليها، فمثلًا، يُعَد toUpperCase تابعًا لسلسلة نصية، ويوضح المثال التالي تابعَيْن يمكنك استخدامهما للتعامل مع المصفوفات:

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

يضيف تابع push قيمًا إلى نهاية مصفوفة ما؛ أما تابع pop فيفعل العكس تمامًا، حيث يحذف القيمة الأخيرة في المصفوفة ويعيدها. وهذه الأسماء السخيفة هي المصطلحات التقليدية للعمليات على المكدِّس stack، والمكدِّس في البرمجة هو أحد هياكل البيانات التي تسمح لك بدفع القيم إليها وإخراجها مرة أخرى بالترتيب المعاكس، بحيث يُبتدأ بإزالة العنصر الذي أضيف آخر مرة، وذلك استنادًا على منطق "آخرهم دخولًا أولهم خروجًا". ولعلك تذكر دالة مكدس الاستدعاءات من الفصل السابق الذي يشرح الفكرة نفسها.

4.5 الكائنات Objects

بالعودة إلى سمير المتحوِّل، فيمكن تمثيل مجموعة من المدخلات اليومية للسجل بمصفوفة، لكن مدخلات التسجيلات تلك فيها أكثر من مجرد عدد أو سلسلة نصية، فكل إدخال يحتاج إلى تخزين قائمة بالأنشطة، وقيمة بولیانية توضح ما إذا كان سمير قد تحول إلى سنجاب أم لا، ونحن نود تجميع ذلك في قيمة واحدة، ثم نضع تلك القيم المجمعة في مصفوفة من مدخلات السجل، وبما أن القيم التي من نوع `object` هي مجرد تجميعات عشوائية من الخصائص، فيمكن إنشاء كائن باستخدام الأقواس في صورة تعبير. انظر كما يلي:

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

لدينا قائمة بالخصائص داخل الأقواس مفصولة بفواصل إنجليزية `,` ولكل خاصية اسم متبوع بنقطتين رأسيتين وقيمة، وحين يُكتب كائن في عدة أسطر، فإن وضع إزاحة بادئة له كما في المثال يجعل قراءته أيسر، والخصائص التي لا تحتوي أسماؤها على أسماء رابطات صالحة أو أرقام صالحة، يجب وضعها داخل علامتي اقتباس. انظر كما يلي:

```
let descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

هذا يعني أن الأقواس لها معنيان في جافاسكربت، فهي تبدأ بكتلة من التعليمات البرمجية إن جاءت في بداية تعليمة ما؛ أما إذا جاءت في موضع آخر، فستصف كائنًا ما. ولعلّ من حسن حظنا أننا نادرًا ما سنحتاج إلى بدء تعليمة بكائن داخل قوسين، لذا لا تشغل بالك كثيرًا بشأن هذا الغموض والإشكال.

كذلك سيعطيك قراءة خاصية غير موجودة القيمة `undefined`، ونستطيع استخدام عامل `=` لإسناد قيمة إلى تعبيرِ خاصية ليغير القيمة الموجودة أصلًا، أو ينشئ خاصيةً جديدةً للكائن إن لم تكن.

بالعودة إلى نموذجنا لمجسات الأخطبوط الذي ذكرناه سابقًا عن الرابطة Binding، فإن رابطات الخصائص متشابهة، فهي تلتقط القيم، لكن قد تكون بعض الرابطات والخصائص الأخرى ممسكة بتلك القيم نفسها، وعليه تستطيع النظر إلى الكائنات على أنها أخطبوطات لها عدد لا نهائي من المجسات، ولكل منها اسم منقوش عليها.

يقطع عامل `delete` أحد المجسات من الأخطبوط السابق، وهذا العامل هو عاملٌ أحادي، كما يحذف الخاصية المسماة من الكائن حين يُطبَّق على خاصيته، وذلك ممكن رغم عدم شيوعه.

```
let anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

عند تطبيق العامل الثنائي `in`على سلسلة نصية وكائن، فسيخبرك إذا كان الكائن به خاصية باسم تلك السلسلة النصية، والفرق بين جعل الخاصية `undefined` وحذفها على الحقيقة، هو أنّ الكائن ما زال يحتفظ بالخاصية في الحالة الأولى مما يعني عدم حمله لقيمة ذات شأن؛ أما في الحالة الثانية فإن الخاصية لم تَعُدْ موجودة، وعليه فستعيد `in` القيمة `false`.

تُستخدَم دالة `Object.keys` لمعرفة الخصائص التي يحتوي عليها الكائن، وذلك بإعطائها كائنًا، فتعيد مصفوفةً من السلاسل النصية التي تمثل أسماء خصائص الكائن. انظر كما يلي:

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

تُستخدَم دالة `Object.assign` لنسخ جميع الخصائص من كائن إلى آخر، انظر كما يلي:

```
let objectA = {a: 1, b: 2};
Object.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

وتكون المصفوفات حينئذ نوعًا من الكائنات المتخصصة في تخزين سلاسل من أشياء بعينها وإذا قيَّمت `typeof []`، فستُنتج "object" وسترى هذه المصفوفات كأخطبوطات طويلة بمجساتها في صف أنيق له عناوين من الأعداد. انظر الآن إلى السجل journal الذي يحتفظ به سمير في صورة مصفوفة من الكائنات:

```
let journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break", "peanuts",
            "juice"],
   squirrel: true},
  /* and so on... */
];
```

4.6 قابلية التغير Mutability

إذا كنت قد قرأت الفصول السابقة، فسترى أنّ أنواع القيم التي تحدثنا عنها من أعداد، وسلاسل نصية، وقيم بوليانية، لا يمكن تغييرها؛ صحيح أنك تستطيع جمعها واستخراج قيم أخرى منها، لكن بمجرد أخذها قيمة لسلسلة نصية فلن تتغير بعدها، وسيبقى النص داخلها كما هو دون تغير، فمثلًا، إن كانت لديك سلسلة نصية تحتوي على "cat"، فلن تستطيع شيفرة أخرى تغيير محرف في هذه السلسلة لتكون "rat".

أما الكائنات فلها شأن آخر، إذ تستطيع تغيير خصائصها، حيث تتخذ قيمة الكائن محتويات مختلفة في كل مرة، كما رأينا قبل قليل أنه يمكن تعديل قيم الكائنات.

حين يكون لدينا عددان 120، و120 فسنقول أنهما نفس العددين سواءً أشارا إلى البتات الحقيقية نفسها أم لا، أما مع الكائنات فهناك فرق بين وجود مرجعين إلى الكائن نفسه، وبين وجود كائنين مختلفين يحتويان نفس الخصائص، انظر الشيفرة التالية:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2);
// → true
```

```
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

تلتقط رابطتي object1 وobject2 الكائن نفسه، لهذا ستتغير قيمة object2 إذا تغير object1، فيقال أنّ لهما "الهوية" نفسها إن صح التعبير؛ أما الرابطة object3، فتشير إلى كائن آخر يحتوي على خصائص object1 نفسها، لكنه منفصل ومستقل بذاته.

قد تكون الروابط نفسها متغيرة أو ثابتة، لكن هذا منفصل عن الطريقة التي تتصرف بها قيمها، ورغم أن القيم العددية لا تتغير، إلا أنك تستطيع استخدام الرابطة let لمتابعة عدد متغير من خلال تغيير القيمة التي تشير الرابطة إليها، وبالمثل، فرغم أن تعريف كائن بالرابطة const سيظل يشير إلى الكائن نفسه ولا يمكن تغييرها لاحقًا، إلا أن محتويات هذا الكائن قابلة للتغيير، كما في المثال التالي:

```
const score = {visitors: 0, home: 0};
// This is okay
score.visitors = 1;
// This isn't allowed
score = {visitors: 1, home: 1};
```

يوازن العامل == بين الكائنات من منظور هويتها، فلا يعطي true إلا إذا كان لكلا الكائنين القيمة نفسها تمامًا؛ أما عند موازنة كائنات مختلفة، فسيعطي false حتى ولو كان لهذه الكائنات الخصائص نفسها، وعليه فليس هناك عملية موازنة "عميقة" في جافاسكربت توازن بين الكائنات من خلال محتوياتها، لكن من الممكن كتابة ذلك بنفسك، وهو موضوع أحد التدريبات في نهاية هذا الفصل (تدريب الموازنة العميقة).

4.7 سجل المستذئب

نعود إلى سمير الذي يظن بأنّه يتحول إلى حيوان في الليل، إذ يبدأ مفسّر جافاسكربت الخاص به، ويضبط البيئة التي يحتاج إليها من أجل سجله journal، انظر كما يأتي:

```
let journal = [];

function addEntry(events, squirrel) {
```

```
    journal.push({events, squirrel});
  }
```

لاحظ أنّ الكائن الذي أضيف إلى السجل يبدو غريبًا نوعًا ما، فبدلًا من التصريح عن الخصائص مثل `events: events` فهو لا يزيد عن إعطاء اسم الخاصية فقط. ويُعَدّ هذا الأسلوب اختصارًا مشيرًا إلى الشيء نفسه، أي إذا كان اسم الخاصية موضوع بين قوسين وليس متبوعًا بقيمة، فستؤخذ قيمته من الرابطة التي تحمل الاسم نفسه؛ لذا، ففي كل ليلة عند العاشرة مساءً -أو في الصباح التالي أحيانًا-، يسجل سمير يومه كالتالي:

```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
          "touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
          "juice"], true);
```

وهو ينوي اتباع أسلوب إحصائي عند حصوله على نقاط بيانات كافية، وذلك لرؤية أيَّ تلك الأحداث هي التي تحث تحوله إلى حيوان ليلًا.

يختلف المتغير في الإحصاء عن المتغير البرمجي، إذ يكون لدينا مجموعة من المقاييس، بحيث يقاس كل متغير بها جميعًا، وتُمثَّل علاقة الترابط Correlation بين المتغيرات بقيمة بين 1- و1، وعلاقة الترابط هي مقياس اعتمادية المتغير الإحصائي على متغير آخر. فإذا كانت قيمة علاقة الترابط هذه صفرًا، فهذا يعني أنّ المتغيرين غير مرتبطان ببعضهما؛ أما إذا كان 1، فهذا يعني أنّ المتغيرين متطابقان تمامًا. بحيث إذا كنت تعرف أحدهما، فأنت تعرف الأخر يقينًا؛ أما إذا كانت قيمة علاقة الترابط تلك 1-، فهذا يعني أنهما متطابقان لكنهما متقابلان، بحيث إن كان الأول true، فالآخر false.

نستخدم معامِل فاي ϕ لحساب مقياس علاقة الترابط بين متغيرين بوليانيين، وهي معادلة يكون دخلها جدول تردد يحتوي على عدد المرات التي لوحظت مجموعات المتغيرات فيها؛ ويصف الخرج علاقة الترابط بينها بحيث يكون عددًا بين 1- و1.

فمثلًا، سنأخذ حدث تناول البيتزا ونضع ذلك في جدول تردد مثل التالي، حيث يشير كل عدد إلى عدد المرات التي وقعت فيها هذه المجموعة في قياساتنا:

9 لا سنجاب، بيتزا	76 لا سنجاب، لا بيتزا
1 سنجاب، بيتزا	4 سنجاب، لا بيتزا

فإذا سمينا هذا الجدول بجدول n مثلًا، فإننا سنستطيع حساب ϕ باستخدام المعادلة التالية:

$$\phi = \frac{n_{11} n_{00} - n_{10} n_{01}}{\sqrt{n_{1.} n_{0.} n_{.1} n_{.0}}}$$

ولا تشغل نفسك بشأن الرياضيات كثيرًا ها هنا، حيث وُضِعت هذه المعادلة لتحويلها إلى جافاسكربت.

تشير الصيغة n_{01} إلى عدد القياسات التي يكون فيها المتغير الأول `squirrel` "السنجاب" غير متحقق أو خطأ false والمتغير الثاني `pizza` "البيتزا" متحقق أو صحيح true ففي جدول البيتزا مثلًا تكون قيمة n_{01} هي 9.

تشير القيمة $n_{1.}$ إلى مجموع القياسات التي كان فيها المتغير الأول متحققًا -أي true- وهي 5 في الجدول المثال. بالمثل، تشير $n_{.0}$ إلى مجموع القياسات التي كان فيها المتغير الثاني يساوي false.

لذا سيكون الجزء الموجود أعلى خط الفصل في جدول البيتزا 1×76−4×9 =40، وسيكون الجزء السفلي هو الجذر التربيعي لـ 5×85×10×80، أو √340000، ونخرج من هذا أن قيمة فاي هي 0.069 تقريبًا، وهي قيمة ضئيلة قطعًا، وعليه فلا يبدو أنّ البيتزا لها تأثير على تحول سمير.

4.8 حساب علاقة الترابط

نستطيع تمثيل جدول من صفين وعمودين في جافاسكربت، باستخدام مصفوفة من أربعة عناصر `[76, 9, 4, 1]`، أو مصفوفة تحتوي على مصفوفتين، بحيث تتكون كل واحدة منهما من عنصرين `[[76, 9], [4, 1]]`، أو كائن له أسماء خصائص، مثل: `"11"` و `"01"`.

غير أنّ المصفوفة المسطحة أسهل وتقصّر طول التعبيرات التي تصل إلى الجدول، وسنفسر فهارس المصفوفة في صورة أعداد ثنائية مكوّنة من بِتَّين، حيث يشير الرقم الأيسر إلى متغير السنجاب، والأيمن إلى متغير الحدث. فمثلًا، يشير العدد الثنائي 10 إلى الحالة التي تحوّل فيها سمير إلى سنجاب، لكن حدث البيتزا مثلًا لم يقع، وقد حدث هذا أربع مرات؛ وبما أن العدد الثنائي 10 ما هو إلا العدد 2 في النظام العشري، فسننخزن هذا الرقم في الفهرس 2 من المصفوفة.

انظر الدالة التي تحسب قيمة معامل ϕ من مثل هذه المصفوفة:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

الشيفرة أعلاه هي ترجمة حرفية لمعادلة فاي الرياضية السابقة في جافاسكربت، وتكون فيها `Math.sqrt` هي دالة الجذر التربيعي التي يوفرها كائن `Math` في بيئة جافاسكربت القياسية، ويجب إضافة حقلين من الجدول لنحصل على حقول مثل n_1. ذلك أن مجموع الصفوف أو الأعمدة لا يُخزَّن في قاعدة بياناتنا مباشرةً وقد احتفظ سمير بسجله لمدة ثلاثة أشهر، وستجد النتائج لتلك الفترة متاحة في صندوق التجارب لهذا الفصل، حيث تُخزَّن في رابطة `JOURNAL` وهي متاحة للتحميل.

سنكرر الآن على كل الإدخالات، وسنسجّل عدد مرات وقوع حدث التحول إلى سنجاب، وذلك لاستخراج جدول بسطرين وعمودين، وذلك كما يلي:

```
function tableFor(event, journal) {
  let table = [0, 0, 0, 0];
  for (let i = 0; i < journal.length; i++) {
    let entry = journal[i], index = 0;
    if (entry.events.includes(event)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
```

```
}

console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

يتحقق تابع `includes` من وجود قيمة ما في المصفوفة، وتَستخدِم الدالة ذلك لتحديد وجود اسم الحدث الذي تريده في قائمة الأحداث في يوم ما.

يبيّن متن الحلقة التكرارية في `tableFor` أيّ صندوق في الجدول يقع فيه إدخال السجل، من خلال النظر في احتواء الإدخال على حدث بعينه أم لا، والنظر هل وقع الحدث مع وقوع التحول السنجابي أم لا، ثم تزيد الحلقة التكرارية الصندوق الصحيح بمقدار 1 داخل الجدول.

لدينا الآن الأدوات التي نحتاج إليها في حساب علاقة الترابط الفردية، والخطوة المتبقية هي إيجاد علاقة الترابط لكل نوع من الأحداث تم تسجيله، وننظر هل سنخرج بنتيجة أم لا.

4.9 حلقات المصفوفات التكرارية

لدينا حلقة تكرارية في دالة `tableFor`، وهي:

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
  // افعل شيئًا هنا بالقيمة
}
```

يكثر هذا النوع من الحلقات في جافاسكربت الكلاسيكية، إذ يشيع المرور على المصفوفات عنصرًا عنصرًا، وذلك باستخدام عداد على طول المصفوفة واختيار كل عنصر على حِدَة، لكن هناك طريقة أبسط لكتابة مثل تلك الحلقات التكرارية في جافاسكربت الحديثة، وذلك كما يلي:

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
}
```

حين تُكتَب حلقة `for` مع الكلمة `of` بعد تعريفِ متغير، ستتكرر على عناصر القيمة المعطاة بعد `of`، ويصلح هذا في المصفوفات، والسلاسل النصية، وبعض هياكل البيانات الأخرى، كما سيأتي بيانه في الفصل السادس.

4.10 التحليل النهائي

نحتاج الآن إلى حساب علاقة الترابط لكل نوع من الأحداث التي وقعت في مجموعة البيانات التي جمعها سمير، ويحتاج هذا الحساب أولًا إلى إيجاد كل نوع من أنواع الأحداث. انظر المثال الآتي:

```
function journalEvents(journal) {
  let events = [];
  for (let entry of journal) {
    for (let event of entry.events) {
      if (!events.includes(event)) {
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", …]
```

تجمع الدالة `journalEvents` كل أنواع الأحداث من خلال المرور على الأحداث وإضافة الغير موجود منها إلى المصفوفة `events`، ونستطيع رؤية كل الالتزامات من خلال الحلقة التالية:

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → carrot:   0.0140970969
// → exercise: 0.0685994341
// → weekend:  0.1371988681
// → bread:   -0.0757554019
// → pudding: -0.0648203724
// and so on...
```

من هذه النتائج، نستطيع القول أن أغلب علاقات الترابط، مثل: أكل الجزر، والخبز، وغيرها، لا تستفز التحول الحيواني لدى سمير لأنها تقترب من الصفر، لكن من ناحية أخرى، فيبدو أنها تزيد في الإجازات الأسبوعية، وعليه سنرشّح النتائج أكثر لنرى أيَّ علاقات الترابط كانت أكبر من 0.1 أو أقل من -0.1.

```
for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
```

```
}
// → weekend:        0.1371988681
// → brushed teeth: -0.3805211953
// → candy:          0.1296407447
// → work:          -0.1371988681
// → spaghetti:      0.2425356250
// → reading:        0.1106828054
// → peanuts:        0.5902679812
```

لدينا عاملان بهما علاقة ترابط أقوى مما سواهما، وهما: أكل الفول السوداني الذي له أثـر إيجـابي قـوي على فرصة التحول، وغسل الأسنان الذي له تأثير قوي كذلك لكن في الاتجاه المعاكس.

دعنا نجرب الشيفرة التالية:

```
for (let entry of JOURNAL) {
  if (entry.events.includes("peanuts") &&
      !entry.events.includes("brushed teeth")) {
    entry.events.push("peanut teeth");
  }
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

هذه نتيجة قوية، إذ تحدث الظاهرة تحديدًا حين يأكل سمير الفول السوداني وينسى غسل أسنانه، وبمـا أنـه عرف هذا، فقد قرر إيقاف أكل الفول السوداني بالكليّة، ووجد ظاهرة تحوله إلى سنجاب لم تتكرر بعدها!

4.11 زيادة على المصفوفات

نريد أن نعرّفك على بعض المفاهيم الأخرى المتعلقة بالكائنات قبل إنهاء هذا الفصـل، فقـد رأينـا في بدايـة هذا الفصل `push` و `pop` لإضافة العناصر وحذفها من نهاية مصفوفة ما؛ أما التابعان الموافقان لإضـافة وحـذف العناصر من بداية المصفوفة، فهما: `unshift` و `shift` وذلك كما يأتي:

```
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
```

```
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

ينظم البرنامج أعلاه مجموعةً من المهام المرتبة في طابور، حيث تضيف مهامًا إلى نهاية الطابور باستدعاء `remember("groceries")` وتَستدعِي `getTask()` إذا أردت فعل شيء ما، وذلك لجلب وحذف العنصر الأمامي من الطابور، كما تضيف دالة `rememberUrgently` مهمةً إلى أول الطابور، وليس إلى آخره.

توفر المصفوفات تابع `indexOf` الذي يبحث في المصفوفة من بدايتها إلى نهايتها عن قيمة معينة، ويعيد فهرس المكان الذي وجد عنده القيمة المطلوبة، وإذا أردت البحث من نهاية المصفوفة بدلًا من بدايتها، فلدينا تابع مماثل اسمه `lastIndexOf`، انظر كما يلي:

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

ويأخذ كلا التابعين `indexOf` و `lastIndexOf` وسيطًا ثانيًا اختياريًا يوضح أين يجب أن يبدأ البحث.

يُعَدّ التابع `slice` من التوابع الأساسية للمصفوفات، إذ يأخذ فهرس البداية والنهاية، ويعيد مصفوفةً تحوي العناصر المحصورة بين هذين الفهرسين، ويكون فهرس البداية موجودًا في هذه المصفوفة الناتجة، أما فهرس النهاية فلا. انظر المثال التالي:

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

إذا لم يُعط فهرس النهاية لتابع `slice`، فسيأخذ كل العناصر التي تلي فهرس البداية، وإن لم تذكر فهرس البداية، فسينسخ المصفوفة كلها.

يُستخدَم تابع `concat` للصق المصفوفات معًا لإنشاء مصفوفة جديدة، وهو في هذا يماثل وظيفة عامل + في السلاسل النصية.

انظر المثال التالي للتابعين السابقين، إذ تأخذ الدالة remove مصفوفةً وفهرسًا، ثم تعيد مصفوفةً جديدةً، بحيث تكون نسخةً من الأصلية بعد حذف العنصر الموجود عند الفهرس المعطى:

```
function remove(array, index) {
  return array.slice(0, index)
    .concat(array.slice(index + 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

إذا مرّرنا وسيطًا ليس بمصفوفة إلى `concat`، فستضاف تلك القيمة إلى المصفوفة الجديدة كما لو كانت مصفوفة من عنصر واحد.

4.12 السلاسل النصية وخصائصها

نستطيع قراءة خصائص من قيم السلاسل النصية، مثل الخاصتين: `length` و `toUpperCase` لكن إذا حاولت إضافة خاصية جديدة، فلن تبقى، انظر المثال التالي:

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

ذلك أن قيم السلاسل النصية، والأعداد، والقيم البوليانية، ليست بكائنات. وعليه فلن تمنعك اللغة من وضع خصائص جديدة على هذه القيم، فهي لا تخزن تلك الخصائص على الحقيقة، إذ لا يمكن تغيير تلك القيم كما ذكرنا من قبل؛ غير أنّ هذه الأنواع لها خصائصها المدمجة فيها، فكل قيمة سلسلة نصية لها عدد من التوابع، لعلّ `slice`، و `indexOf` أكثرها نفعًا واستخدامًا، واللذين يشبهان في وظائفهما التابعَين المذكورَين قبل قليل، انظر المثال التالي:

```
console.log("coconuts".slice(4, 7));
// → nut
console.log("coconut".indexOf("u"));
// → 5
```

الفرق بينهما أنه يستطيع تابع `indexOf` في السلسلة النصية، البحث عن سلسلة تحتوي على أكثر من محرف واحد؛ بينما تابع المصفوفة الذي يحمل الاسم نفسه لا يبحث إلا عن عنصر واحد، أي كما في المثال التالي:

```
console.log("one two three".indexOf("ee"));
// → 11
```

يحذف تابع `trim` المسافات البيضاء، مثل: المسافات، والأسطر الجديدة، وإزاحات الجداول، وما شابه ذلك، من بداية ونهاية السلسلة النصية، ومثال على ذلك:

```
console.log("  okay \n ".trim());
// → okay
```

الدالة `zeroPad` المستخدَمة في الفصل السابق، موجودة هنا على أساس تابع أيضًا، ويسمى `padStart`، حيث يأخذ الطول المطلوب، ومحرف الحشو على أساس وسائط، كما في المثال التالي:

```
console.log(String(6).padStart(3, "0"));
// → 006
```

تستطيع تقسيم سلسلة نصية عند كل ظهور لسلسلة أخرى باستخدام تابع `split`، ثم دمجها مرةً أخرى باستخدام تابع `join` أي كما في المثال التالي:

```
let sentence = "Secretarybirds specialize in stomping";
let words = sentence.split(" ");
console.log(words);
// → ["Secretarybirds", "specialize", "in", "stomping"]
console.log(words.join(". "));
// → Secretarybirds. specialize. in. stomping
```

يمكن تكرار السلسلة النصية باستخدام تابع `repeat`، حيث ينشِئ سلسلةً نصيةً جديدةً تحتوي نسخًا متعددةً من السلسلة الأصلية، وملصقةً معًا. انظر المثال التالي:

```
console.log("LA".repeat(3));
// → LALALA
```

وبالنسبة لخاصية `length` للسلاسل النصية التي رأيناها من قبل، فيحاكي الوصول إلى المحرف داخل سلسلة نصية، الوصول إلى عناصر المصفوفة مع فارق بسيط سنناقشه في الفصل الخامس. انظر المثال التالي:

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

4.13 معامل الباقي rest

من المفيد لدالة قبول أي عدد من الوسائط، فمثلًا، تحسب الدالة `Math.max` القيمة العظمى لكل الوسائط المعطاة؛ إذ يمكننا تحقيق ذلك بوضع ثلاث نقاط قبل آخر معامِل للدالة، كما يلي:

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

وحين تُستدعى هذه الدالة فإن معامل الباقي rest يكون ملزَمًا بمصفوفة تحتوي كل الوسائط الأخرى، وإذا كان ثمة معامِلات أخرى قبله، فلا تكون قيمها جزءًا من المصفوفة؛ أما حين يكون هو المعامل الوحيد كما في حالة `max`، فستحمل المصفوفة كل الوسائط.

تستطيع استخدام صيغة النقاط الثلاثة لاستدعاء دالة مع مصفوفة وسائط، كما في المثال التالي:

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

يوسع هذا المصفوفة إلى استدعاء الدالة ممررًا عناصرها على أساس وسائط منفصلة، ومن الممكن إضافة مصفوفة مثل هذه إلى جانب وسائط أخرى كما في `max(9, ...numbers, 2)`، كذلك تسمح صيغة الأقواس المربعة لمصفوفة، لعامل النقاط الثلاثة، بتوسيع مصفوفة أخرى داخل هذه المصفوفة الجديدة، كما في المثال التالي:

```
let words = ["never", "fully"];
console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

4.14 الكائن Math

كما رأينا سابقًا، فـ `Math` ما هو إلا حقيبةٌ من دوال التعامل مع الأعداد، مثل: `Math.max` للقيمة العظمى، و`Math.min` للقيمة الصغرى، و`Math.sqrt` للجذر التربيعي.

يُستخدَم كائن Math على أساس حاوية لمجموعة من الوظائف المرتبطة ببعضها بعضًا، كما أنّه كائن وحيد، إذ لا يوجد كائن آخر يحمل الاسم نفسه، وهو غير مفيد أيضًا إن جاء على أساس قيمة، فهو يوفر فضاء اسم namespace لئلا تضطر الدوال والقيم لتكوّن روابط عامة global bindings، حيث تلوث كثرة هذه الروابط العامة فضاء الاسم، فكلما زاد عدد الأسماء المحجوزة زادت فرصة تغيير قيمة رابطة حالية بالخطأ، ولا بأس مثلًا بتسمية شيء ما باسم max في برنامج تكتبه، إذ أنّ دالة max المدمجة بجافاسكربت محفوظة بأمان داخل كائن Math، لذا فلن تتغير بفعل منك.

لن توقفك أو تحذرك جافاسكربت -على عكس كثير من اللغات الأخرى- من تعريف رابطة باسم مأخوذ من قبل، إلا أن تكون رابطةً صرحْتَ عنها باستخدام let، أو const، أما الروابط القياسية أوالمصرَّح عنها باستخدام var، أو function فلا.

ستحتاج إلى كائن Math إن أردت تنفيذ بعض العمليات المتعلِّقة بحساب المثلثات، وذلك لاحتوائه على دوال الجيب sin، وجيب التمام cos، والظل tan، إضافةً إلى دوالها المعكوسة، وهي: asin، وacos، وatan، كما أن العدد باي π متاح أيضًا في جافاسكربت في صورة Math.PI؛ وكُتِبت بحروف إنجليزية كبيرة تطبيقًا لعادة قديمة في البرمجة، إذ تُكتَب أسماء القيم الثابتة بالحروف الكبيرة.

```
function randomPointOnCircle(radius) {
  let angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

ولا تقلق إن لم تكن قد تعرضت لهذه الدوال من قبل، حيث ستُشرح حين يأتي ذكرها في الفصل الرابع عشر من الكتاب، وقد استخدمنا الدالة Math.random في المثال أعلاه، إذ تعيد عددًا عشوائيًا وهميًا بين الصفر والواحد في كل مرة تستدعيها، مع استثناء الواحد نفسه فلا تعيده. انظر المثال التالي:

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

رغم أنّ الحواسيب آلات تعيينية، أي تتصرف بالطريقة نفسها إن أعطيتها المدخلات ذاتها، إلا أنّه من الممكن جعلها تنتج أعدادًا قد تبدو عشوائية، ولفعل ذلك تحتفظ الآلة بقيمة مخفيّة، وتُجري حسابات معقدة

على هذه القيمة المخفيّة لإنشاء واحدة جديدة في كل مرة تسألها فيها إعطاءك عددًا عشوائيًا؛ كما تخزِّن القيمة الجديدة وتعيد عددًا مشتقًا منها، وهكذا تستطيع إنتاج أعداد بطريقة تبدو عشوائية ويصعب التنبؤ بها؛ أما إن أردت أعدادًا صحيحةً وعشوائيةً بدلًا من الأعداد الكسرية، فاستخدام `Math.floor` على الخرج الذي تحصل عليه من `Math.random`، حيث تقرِّب العدد إلى أقرب عدد صحيح.

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

سيعطيك ضرب العدد العشوائي في 10 عددًا أكبر من أو يساوي الصفر، وأصغر من العشرة؛ وبما أن `Math.floor` تقرِّبه، فسينتج هذا التعبير أعدادًا من 0 إلى 9 باحتمالات متساوية.

تقرِّب الدالة `Math.ceil` إلى عدد صحيح، كما تقرِّب الدالة `Math.round` إلى أقرب رقم صحيح، وتأخذ الدالة `Math.abs` القيمة المطلقة لعدد ما، أي تنفي القيمة السالبة وتترك القيمة الموجبة كما هي.

4.15 التفكيك

انظر الشيفرة التالية:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}
```

إذا عدنا إلى دالة `phi` السابقة، فإن أحد الأسباب التي يجعل هذه الدالة صعبةً في قراءتها، هو أنّه لدينا رابطة تشير إلى مصفوفتنا، بينما نريد رابطات لعناصر المصفوفة، أي `let n00 = table[0]`، وهكذا.

لحسن الحظ لدينا طريقة مختصرة في جافاسكربت تفعل ذلك:

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

هذا يصلح أيضًا للرابطات التي أنشئت باستخدام `let` و`var` و`const`، فإذا كانت القيمة التي تربطها مصفوفةً تستطيع استخدام الأقواس المربعة لتنظر داخلها لتربط محتوياتها.

كذلك بالنسبة للكائنات إذ نستخدم الأقواس العادية بدلًا من المربعة، انظر المثال التالي:

```
let {name} = {name: "Faraji", age: 23};
console.log(name);
// → Faraji
```

لاحظ أنك إذا حاولت تفكيك `null`، أو `undefined`، فستحصل على خطأ، وكذلك إن حاولت الوصول مباشرةً إلى إحدى خصائص هاتين القيمتين.

4.16 صيغة JSON

بما أن الخصائص تلتقط قيمها ولا تحتويها، فتُخزَّن المصفوفات والكائنات في ذاكرة الحاسوب على أساس سلاسل من البتات حاملةً عناوينًا لمحتوياتها، وهذه العناوين هي أماكن في الذاكرة، لذا فإن احتوت مصفوفة على مصفوفة أخرى داخلها، فستشغل من الذاكرة قطاعًا واحدًا على الأقل للمصفوفة الداخلية، وواحدًا آخرًا للمصفوفة الخارجية، حيث سيحتوي على عدد ثنائي يمثل موضع المصفوفة الداخلية، مع أشياء أخرى قطعًا.

وإن أردت حفظ بيانات في ملف لاستخدامها لاحقًا أو إرسالها إلى حاسوب آخر عبر الشبكة، فعليك تحويل هذه العناوين المتشابكة إلى وصف يمكن تخزينه أو إرساله، وبهذا تستطيع إرسال ذاكرة حاسوبك كلها مع عنوان القيمة التي تريدها، رغم أن هناك طرق أفضل لهذا.

والذي نستطيع فعله هو تحويل سَلسلة البيانات، بمعنى تحويلها إلى وصف بسيط وثَمَّ استخدام طريقة مشهورة تسمى JSON -تنطق جَيسون- وهي اختصار لصيغة الكائنات في جافاسكربت JavaScipt Object Notation، وتُستخدَم استخدامًا واسعًا على أساس طريقة لتخزين البيانات، وصيغة للتواصل في الإنترنت حتى في اللغات الأخرى غير جافاسكربت.

تحاكي صيغة JSON طريقة جافاسكربت في كتابة المصفوفات والكائنات مع بعض القيود، لذا فيجب إحاطة كل أسماء الخصائص بعلاماتي تنصيص مزدوجة، ولا يُسمح إلا بتعابير البيانات البسيطة، فلا استدعاءات لدوال، ولا روابط، ولا أي شيء فيه حوسبة حقيقية؛ والتعليقات ممنوعة أيضًا.

وإذا عدنا -مرةً أخرى- إلى مثال سمير المتحوِّل، وأردنا تمثيل مدخلًا للسجل الذي يحتفظ به في صورة بيانات JSON، فسيبدو هكذا:

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

تعطينا جافاسكربت دالتي `JSON.stringify`، و `JSON.parse` لتحويل البيانات من وإلى هذه الصيغة، فالأولى تأخذ قيمة من جافاسكربت، وتعيد سلسلةً نصيةً مرمّزةً بصيغة JSON، والثانية تأخذ هذه السلسلة وتحولها إلى القيمة التي ترمِّزها، كما في المثال التالي:

```
let string = JSON.stringify({squirrel: false,
                            events: ["weekend"]});
console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

4.17 خاتمة

توفر الكائنات والمصفوفات طريقًا لتجميع عدة قيم ووضعها في قيمة واحدة، ويسمح هذا نظريًا لنا بوضع بعض الأشياء المرتبطة ببعضها في حقيبة واحدة، وذلك لنتعامل مع الحقيبة كلها بدلًا من محاولة الإمساك بهذه الأشياء واحدةً واحدةً بأيدينا.

تملك أغلب القيم في جافاسكربت خصائص، باستثناء: `null`، و`undefined`، ونستطيع الوصول إلى تلك الخصائص باستخدام `value.prop`، أو `value["prop"]`.

تميل الكائنات إلى استخدام أسماء خصائصها وتخزين مجموعة منها؛ أما المصفوفات فتحتوي غالبًا على كميات مختلفة من القيم المتطابقة نظريًا وتستخدم الأعداد (بدءًا من الصفر) على أساس أسماء لخصائصها.

لدينا بعض الخصائص المسماة في المصفوفات مثل `length`، وعددًا من التوابع التي هي دوال تعيش داخل الخصائص وتتحكم عادةً في القيم التي تكون جزءًا منها.

تستطيع تطبيق التكرار على المصفوفات باستخدام نوع خاص من حلقة `for` التكرارية أي:

```
for(let element of array)
```

4.18 تدريبات

4.18.1 مجموع مجال ما

لقد أشرنا في مقدمة هذا الكتاب إلى ما يلي على أنه طريقة لطيفة لحساب مجموع مجال ما من الأعداد:

```
console.log(sum(range(1, 10)));
```

اكتب دالة range تأخذ وسيطين، هما: start، وend، وتعيد مصفوفةً تحتوي على جميع الأعداد من start حتى end مضمِّنًا العنصر end فيها، ثم اكتب دالة sum تأخذ مصفوفة من الأرقام وتعيد مجموعها، وشغِّل هذا البرنامج وانظر هل يعيد 55 أم لا.

وفي تدريب إضافي، عدِّل دالة range أعلاه لتأخذ وسيطًا ثالثًا اختياريًا يوضح قيمة الخطوة المستخدمة عند بناء المصفوفة، وإن لم يكن ثمة خطوات معطاة فإن العناصر تزيد بمقدار واحد متوافقةً مع السلوك الأول.

ويجب إعادة [1, 3, 5, 7, 9] عند استدعاء الدالة range(1, 10, 2)، كما يجب عليك التأكد من كونه يعمل مع قيم الخطوات السالبة، بحيث سنحصل على المصفوفة [5, 4, 3, 2] من استدعاء range(5, 2, -1).

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// اكتب شيفرتك هنا.

console.log(range(1, 10));
// → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log(range(5, 2, -1));
// → [5, 4, 3, 2]
console.log(sum(range(1, 10)));
// → 55
```

إرشادات الحل

يُعَدّ بناء مصفوفة أمرًا يسير إذا ابتدأنا رابطةً إلى [] -مصفوفة فارغة جديدة-، واستدعاء لتابعها push لإضافة قيمة ما، كما لا تنس إعادة المصفوفة عند نهاية الدالة، وبما أن الحد النهائي مضمَّن، فستحتاج إلى استخدام عامل <= بدلًا من < للتحقق من نهاية الحلقة التكرارية.

قد يكون معامل الخطوة step parameter معامِلًا اختياريًا، ويكون الافتراضي له 1 باستخدام عامل =. كما تُنفَّذ الخطوة السالبة في range بكتابة حلقتين تكراريتين منفصلتين، إحداهما للعد التصاعدي، والأخرى للعد التنازلي، إذ تحتاج الموازنة التي تتحقق من نهاية الحلقة، إلى >= بدلًا من <= عند العد التنازلي.

قد يكون من المفيد أحيانًا استخدام خطوة افتراضية مختلفة، -1 تحديدًا، حين تكون نهاية المجال أصغر من البداية، وهكذا تعيد range(5, 2) شيئًا مفيدًا بدلًا من أن نعلق داخل حلقة لا نهائية، كذلك من الممكن الإشارة إلى معامِلات أخرى في القيمة الافتراضية للمعامِل.

4.18.2 عكس مصفوفة

تملك المصفوفات تابع `reverse`، إذ يغيرها بعكس الترتيب الذي تظهر عناصرها به؛ اكتب في هذا التدريب الدالتين `reverseArray`، و`reverseArrayInPlace`، بحيث تأخذ الأولى مصفوفةً على أساس وسيط لها وتنتج مصفوفة جديدة بها العناصر نفسها ولكن بترتيب معكوس؛ أما الثانية فتتصرف مثل تابع `reverse`، إذ تعدّل المصفوفة المعطاة على أساس وسيط بعكس عناصرها؛ لا تستخدم تابع `reverse` القياسي في أي من الدالتين.

على ضوء الملاحظات التي ذكرتها في الفصل السابق حول الآثار الجانبية والدوال النقية، أيّ صورة برأيك تتوقع أنها ستكون مفيدةً في مواقف أكثر؟ وأي واحدة ستكون أسرع؟

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// اكتب شيفرتك هنا.

console.log(reverseArray(["A", "B", "C"]));
// → ["C", "B", "A"];
let arrayValue = [1, 2, 3, 4, 5];
reverseArrayInPlace(arrayValue);
console.log(arrayValue);
// → [5, 4, 3, 2, 1]
```

إرشادات الحل

هناك طريقتان واضحتان لتنفيذ `reverseArray`، أولاهما المرور على مصفوفة الدخل من الأمام إلى الخلف، واستخدام التابع `unshift` على المصفوفة الجديدة لإدخال كل عنصر في بدايتها؛ أما الثانية، فهي تطبيق تكرار خلفي على مصفوفة الدخل، واستخدام التابع `push`، وسيتطلب ذلك مواصفات خاصة لـ `for` مثل:

```
(let i = array.length - 1; i >= 0; i--)
```

سيكون عكس المصفوفة في مكانها أصعب، وعليك أن تكون أكثر حذرًا لئلا تكتب فوق العناصر التي ستحتاج إليها لاحقًا، وإذا استخدمت `reverseArray`، أو نسخت المصفوفة كلها باستخدام `array.slice(0)` فسينجح ذلك، لكننا نرى هذا غشًا، فالفكرة هي تبديل العنصرين الأول والأخير، ثم الثاني من البداية والثاني من النهاية، وهكذا، ويمكنك فعل ذلك بالتكرار على نصف طول المصفوفة.

استخدم `Math.floor` للتقريب -لن تحتاج إلى التعامل مع العنصر الأوسط في مصفوفة عدد عناصرها فردي-، ثم بدّل بين العنصر في الموضع `i` مع العنصر في الموضع `array.length - 1 - i`.

يمكنك استخدام رابطة محلية لإمساك أحد العناصر ثم الكتابة فوقه بصورته المعكوسة mirror image، ثم وضع القيمة التي من الرابطة المحلية في الموضع الذي كانت فيه الصورة المعكوسة.

4.18.3 قائمة

يمكن استخدام الكائنات مثل كتل بها قيم من أجل بناء أي نوع من هياكل البيانات، وأكثر هذه الأنواع شهرة هي القائمة -يختلف هذا عن المصفوفات-، وهي مجموعة متشعبة من الكائنات، يحمل أولها مرجعًا إلى الكائن الثاني، والثاني إلى الثالث، وهكذا.

```
let list = {
  value: 1,
  rest: {
    value: 2,

    rest: {      value: 3,
      rest: null
    }
  }
};
```

وتكون الكائنات الناتجة في صورة سلسلة كما يلي:

والجميل في هذه القوائم أنها تستطيع مشاركة أجزاء من بنيتها، فإذا أنشأنا قيمتين جديدتين، مثل: `{value: 0, rest: list}`، و `{value: -1, rest: list}`، بحيث تشير `list` إلى الرابطة المصرح عنها من قبل، فستكون كل واحدة منهما قائمةً مستقلةً لكنهما يتشاركان البنية التي تكوّن آخر ثلاثة عناصر فيهما، ولا تزال القائمة الأصلية قائمةً من ثلاثة عناصر أيضًا.

اكتب دالة `arrayToList` التي تكون بنية قائمة مثل التي استعرضناها إذا أُعطيت الوسيط `[1, 2, 3]`، وكذلك دالة `listToArray` التي تنتج دالة من قائمة ما، ثم أضف دالة `prepend` المساعِدة التي تأخذ عنصرًا وقائمةً وتنشئ قائمةً جديدةً تضيف العنصر إلى أول قائمة المدخلات، ودالة `nth` التي تأخذ قائمةً وعددًا

وتعيد العنصر عند الموضع المعطى في القائمة -مع صفر يشير إلى العنصر الأول-، أو undefined إذا لم يكن ثمة عنصر من هذا النوع، واكتب كذلك نسخة تعاودية من nth إن لم تكن قد فعلت.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// اكتب شيفرتك هنا.

console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

إرشادات الحل

يكون بناء قائمة أسهل حين نبدأ من الخلف إلى الأمام، لذا قد تكرِّر arrayToList على المصفوفة خلفيًا لكل عنصر -انظر التدريب السابق-، وأضف كائنًا إلى القائمة.

تستطيع استخدام رابطة محلية لتمسك بجزء القائمة الذي تم بناؤه، بالشكل:

```
list = {value: X, rest: list}
```

لإضافة العنصر؛ وللمرور على قائمة -في listToArray وnth- فيمكن استخدام حلقة for مثل هذه:

```
for (let node = list; node; node = node.rest) {}
```

إن كيفية العمل هنا هي أنه في كل تكرار للحلقة، تشير node إلى القائمة الفرعية الحالية، ويستطيع المتن قراءة خاصية value له للحصول على العنصر الحالي، ثم تنتقل node إلى العنصر التالي في القائمة الفرعية في نهاية التكرار؛ أما إذا حصلنا على null، فنكون قد وصلنا إلى نهاية القائمة، وتنتهي الحلقة التكرارية.

ستنظر النسخة التعاودية من nth إلى جزء أصغر من ذيل القائمة، وتَعُدّ تنازليًا حتى تصل إلى الفهرس 0، والتي تعيد خاصية value عندئذ لنفس العقدة التي تنظر إليها. ولكي تحصل على العنصر رقم صفر في القائمة، فستأخذ خاصية value لعقدتها الرئيسية head node؛ أما إذا أردت الحصول على العنصر رقم N+1، فستأخذ العنصر رقم N من القائمة الذي يكون في خاصية rest في تلك القائمة.

4.18.4 الموازنة العميقة

يوازن العامل `==` بين كائنين من حيث هويتهما، لكن قد تفضل أحيانًا موازنة القيم من حيث خصائصها الحقيقية.

اكتب دالة `deepEqual` بحيث تأخذ قيمتين وتعيد `true` فقط إن كان لهما القيمة نفسها، أو كانت الكائنات لها الخصائص نفسها، بحيث تكون قيم الخصائص متساوية إذا ووزِنت مع استدعاء `deepEqual` تعاوديًا، ولمعرفة إن كان يجب موازنة القيم مباشرةً باستخدام عامل `===`، أو توازن خصائصها، فتستطيع استخدام عامل `typeof`، فإذا أخرَجَت `"object"` لكلا القيمتين، فيجب تنفيذ الموازنة العميقة، لكن يجب العلم أنّ `typeof null` تنتج `"object"` كذلك، وهذا استثناء بسبب حادثة قديمة وقعت، لكن ضعه في حسابك على أي حال كاستثناء.

ستكون دالة `Object.keys` مفيدةً لك حين تريد المرور على خصائص الكائنات للموازنة بينها.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// اكتب شيفرتك هنا.

let obj = {here: {is: "an"}, object: 2};
console.log(deepEqual(obj, obj));
// → true
console.log(deepEqual(obj, {here: 1, object: 2}));
// → false
console.log(deepEqual(obj, {here: {is: "an"}, object: 2}));
// → true
```

إرشادات الحل

إذا أردت معرفة إذا كنت تتعامل مع كائن حقيقي أم لا، فستستخدم شيئًا مثل هذا:

```
typeof x == "object" && x != null
```

لكن لا توازن الخصائص إلا حين يكون الوسيطان كائنان، أما في بقية الحالات، فيمكنك إعادة نتيجة تطبيق `===` مباشرةً.

استخدم `Object.keys` للمرور على الخصائص، وستحتاج إلى رؤية ما إذا كان كلا الكائنين لهما نفس مجموعة أسماء الخصائص أم لا، وما إذا كانت تلك الخصائص لها قيم متطابقة أيضًا أم لا.

وإحدى الطرق التي تفعل بها ذلك هي التأكد من أن كلا الكائنين لهما العدد ذاته من الخصائص -أطوال قوائم الخصائص متساوية-، بعدها إذا كررت على خصائص أحد الكائنات للموازنة بينها، فتأكد أولًا أنّ الكائن الآخر له خاصية بالاسم نفسه، فإذا كان لديهما العدد نفسه من الخصائص وكانت كل الخصائص في أحدهما موجودة في الآخر، فسيكون لهما المجموعة ذاتها من أسماء الخصائص.

وأفضل طريقة لإعادة القيمة الصحيحة من دالة، هي بإعادة false في حالة عدم التطابق، وإعادة true في نهاية الدالة.

دورة تطوير تطبيقات الويب
باستخدام لغة PHP

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

5. الدوال العليا higher-order functions

> قال تزو-لي متبجحًا لتزو-سو أنّ برنامجه الأخير يحتوي على مائتي ألف سطر برمجي خلاف التعليقات، فرد عليه تزو-سو بأن برنامجه به حوالي مليون سطر برمجي! حينئذ قال السيد يوان-ما أنّ أفضل برامجه ليس به سوى خمس مئة سطر فقط، فلما سمعا ذلك عرفا أنّ قولهما ليس بشيء.
>
> — يوان-ما Yuan-Ma، كتاب البرمجة The Book Of Programming

> هناك طريقتان لتصميم البرمجيات، تتمثل إحداهما في جعل البرنامج سهلًا جدًا، بحيث لا تجد فيه أي عيوب، والأخرى بجعله معقدًا جدًا بحيث لا تستطيع رؤية عيوب ظاهرة.
>
> — توني هور C. A. R. Hoare، محاضرة جائزة تيورنج، 1980 في رابطة الآلات البرمجية ACM.

تتكلف البرامج بموارد أكثر كلما زاد حجمها، وذلك ليس بسبب الوقت الذي تستغرقه من أجل بنائها، بل لأنّ الحجم الكبير يتبعه تعقيد أكثر، ويحيّر ذلك التعقيد المبرمجين العاملين عليه، حيث تجعلهم تلك الحيرة يرتكبون أخطاءً في صورة زلات برمجية Bugs، وعليه يكون البرنامج الكبير فرصةً كبيرةً لهذه الزلات بالاختفاء وسط الشيفرات، مما يصعّب الوصول إليها.

لنَعُدْ إلى المثالين الأخيرَين المذكورَين في مقدمة الكتاب، حيث يحتوي المثال الأول منهما على ستة أسطر، وهو مستقل بذاته، انظر كما يلي:

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
```

```
  count += 1;
}
console.log(total);
```

أما الثاني فيعتمد على دالتين خارجيتين، ويتكوّن من سطر واحد فقط، كما يلي:

```
console.log(sum(range(1, 10)));
```

برأيك، أيهما أكثر عرضةً لتكون فيه زلة برمجية؟

إذا حسبنا حجم تعريفي `sum` و `range`، فسيكون البرنامج الثاني أكبر من الأول، لكن لا زلنا نراه أكثر صِحة، حيث عبَّر عن الحل بألفاظ تتوافق مع المشكلة المحلولة، فلا يتعلق استدعاء مجال من الأعداد بالحلقات التكرارية والعدادات بقدر ما يتعلق بالمجالات والمجموع الإجمالي؛ وتحتوي تعاريف هذه الألفاظ (دالتي `sum` و `range`) على حلقات تكرارية، وعدادات، وتفاصيل أخرى، وبما أنها تعبر عن مفاهيم أبسط من البرنامج ككل، فهي أدنى ألا تحتوي على أخطاء.

5.1 التجريد Abstractions

تسمى هذه الأنواع من الألفاظ في السياقات البرمجية بالتجريدات Abstractions، وهي تخفي التفاصيل الدقيقة وتعطينا القدرة على الحديث عن المشاكل على مستوى أعلى أو أكثر تجريدًا. انظر هاتين الوصفتين أدناه لتقريب الأمر:

- الوصفة الأولى:

> ضع 1 كوب من البازلاء لكل شخص في حاوية. أضف الماء حتى تغطي البازلاء. اترك البازلاء في الماء لاثنتي عشرة ساعة على الأقل. أخرج البازلاء من الماء وضعها في إناء الطهي. أضف 4 أكواب من الماء لكل شخص. ضع الغطاء على الإناء واترك البازلاء على نار هادئة جدًا لمدة ساعتين. خذ نصف بصلة لكل شخص وقطعها إلى قطع صغيرة باستخدام سكين. أضف البصل المقطع إلى البازلاء. خذ جزرة لكل شخص. قطع الجزر إلى قطع صغيرة باستخدام سكين. أضف الجزر إلى البازلاء. اترك ذلك كله على النار لعشرة دقائق أخرى.

- الوصفة الثانية:

> جهز لكل شخص 1 كوب من البازلاء المجففة، ونصف بصلة مقطعة، وساق من الكرفس وجزرة.
>
> انقع البازلاء في الماء لمدة 12 ساعة، واطهها على نار هادئة جدًا لساعتين مع مقدار 4 أكواب من الماء، وبعدها قطع الخضروات وأضفها، ثم اترك ذلك كله لعشرة دقائق إضافية.

لا شك أن الوصفة الثانية أقصر وأيسر في التفسير والفهم، لكن ستحتاج إلى فهم المصطلحات الخاصة بالطهي، مثل: النقع، والطهي، والتقطيع، والتجفيف (هذه المصطلحات للتقريب مثالًا، وشاهدها أن تكون على دراية بمصطلحات مجال المشكلة التي تريد حلها، وإلا فهي معروفة لكل أحد).

يقع الكثير من المبرمجين في خطأ الوصفة الأولى عند سردهم للخطوات الدقيقة والصغيرة التي على الحاسوب تنفيذها خطوةً بخطوة، وذلك بسبب عدم ملاحظتهم للمفاهيم العليا في المشكلة التي بين أيديهم، ويُعَدّ الانتباه عند حلك لمشكلة بهذا الأسلوب مهارةً مفيدةً جدًا.

5.2 تجريد التكرار

تُعَدّ الدوال البسيطة طريقة ممتازة لبناء تجريدات، غير أنها تعجز عن ذلك أحيانًا، فمن الشائع أن يفعل البرنامج شيئًا ما بعدد معيّن من المرات باستخدام حلقة `for`، فمثلًا:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

فهل نستطيع تجريد مفهوم "فعل شيء ما عددًا من المرات قدره N" في صورة دالة؟ بدايةً، من السهل كتابة دالة تستدعي `console.log` عددًا من المرات قدره N:

```
function repeatLog(n) {
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
}
```

لكن ماذا لو أردنا فعل شيء آخر غير تسجيل الأعداد؟ بما أنّه يمكن تمثيل "فعل شيء ما" على أساس دالة، والدوال ما هي إلا قيم، فسنستطيع تمرير إجراءنا على أساس قيمة دالة، وذلك كما يلي:

```
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}

repeat(3, console.log);
// → 0
// → 1
```

```
// → 2
```

لسنا في حاجة إلى تمرير دالة معرَّفة مسبقًا إلى repeat، فغالبًا من السهل إنشاء قيمة دالة عند الحاجة.

```
let labels = [];
repeat(5, i => {
  labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

وكما ترى، فهذا مهيكل في صورة محاكية لحلقة for، إذ يصف نوع الحلقة التكرارية أولًا، ثم يوفر متنًا لها، غير أنّ المتن الآن مكتوب على أساس قيمة دالة مغلّفة بأقواس الاستدعاء إلى repeat، وهذا هو السبب الذي يجعل من الواجب إغلاقها بقوس إغلاق معقوص } وقوس إغلاق عادي) بأقواس إغلاق، وفي حالة هذا المثال عندما يكون المتن تعبيرًا واحدًا وصغيرًا، فيمكنك إهمال الأقواس المعقوصة وكتابة الحلقة في سطر واحد.

5.3 الدوال العليا

تسمى الدوال التي تعمل على دوال أخرى سواءً بأخذها على أساس وسائط أو بإعادتها لها، باسم الدوال العليا higher-order functions، وبما أنَّ الدوال ما هي إلا قيم منتظمة، فلا شيء جديد في وجود هذا النوع منها، والمصطلح قادم من الرياضيات حين يؤخذ الفرق بين الدوال والقيم الأخرى على محمل الجد.

وتسمح لنا الدوال العليا بعملية التجريد على القيم والإجراءات أيضًا، وتأتي في عدة أشكال وصور، فقد تنشِئ الدوال دوالًا أخرى جديدةً، كما يلي:

```
function greaterThan(n) {
  return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

وقد تُغيِّر الدوال دوالًا أخرى، كما في المثال التالي:

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    let result = f(...args);
```

```
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

كما توفر الدوال أنواعًا جديدةً من تدفق التحكم control flow:

```
function unless(test, then) {
  if (!test) then();
}

repeat(3, n => {
  unless(n % 2 == 1, () => {
    console.log(n, "is even");
  });
});
// → 0 is even
// → 2 is even
```

يوفر تابع مصفوفة مدمج `forEach` شيئًا مثل حلقة `for/of` التكرارية على أساس دالة عليا، وذلك كما يلي:

```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

5.4 مجموعات البيانات النصية

تُعَدّ معالجة البيانات إحدى الجوانب التي تبرز فيها أهمية الدوال العليا، ولضرْب مثال على ذلك سنحتاج إلى بعض البيانات الفعلية، كما سنستخدم في هذا الفصل مجموعة بيانات عن نصوص وأنظمة كتابة، مثل: اللاتينية، والعربية، والسيريلية (حروف اللغات الأوراسية، مثل: الروسية، والبلغارية).

ترتبط أغلب محارف اللغات المكتوبة بنص معين، ولعلك تذكر حديثنا عن الترميز الموحد Unicode الذي يسند عددًا لكل محرف من محارف هذه اللغات، حيث يحتوي هذا المعيار على 140 نصًا مختلفًا، من بينها 81 لا

تزال مستخدمةً، في حين صارت 59 منها مهملةً أو تاريخيةً، أي لم تَعُدْ مستخدمة؛ فعلى سبيل المثال، انظر هذه الكتابة من اللغة التاميلية:

تحتوي مجموعة البيانات على بعض أجزاء البيانات من النصوص المئة والأربعين المعرَّفة في اليونيكود، وهي متاحة في صندوق اختبار هذا الفصل على هيئة رابطة SCRIPTS.

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

يخبرنا الكائن السابق بكل من: اسم النص، ومجالات اليونيكود المسنَدة إليه، واتجاه الكتابة، والزمن التقريبي لنشأة هذه اللغة، وما إذا كانت مستخدمةً أم لا، ورابط إلى مزيد من التفاصيل والبيانات عنها؛ وقد يكون اتجاه الكتابة من اليسار إلى اليمين "ltr"، أو من اليمين إلى اليسار "rtl"، كما في حالة اللغتين العربية والعبرية، أو من الأعلى إلى الأسفل "ttb" كما في حالة اللغة المنغولية.

تحتوي خاصية ranges على مصفوفة من مجالات المحارف، حيث يكون كل منها مصفوفةً من عنصرين، هما الحدين الأدنى والأعلى، ويُسنَد أيّ رمز للمحارف بين هذه المجالات إلى النص، كما يُضمَّن الحد الأدنى فيها؛ أما الحد الأعلى فلا، أي يُعَدّ رمز 994 محرفًا قبطيًا Coptic في المثال السابق؛ أما الرمز 1008 فلا.

5.5 ترشيح المصفوفات

نستخدم دالة filter لإيجاد النصوص واللغات التي ما زالت مستخدَمةً في مجموعات البيانات، إذ تُرشِّح عناصر المصفوفة التي لا تجتاز اختبارًا تجريه عليها:

```
function filter(array, test) {
  let passed = [];
  for (let element of array) {
```

```
    if (test(element)) {
      passed.push(element);
    }
  }
  return passed;
}

console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", …}, …]
```

تستخدم الدالة وسيطًا اسمه `test`، وهو قيمة دالةٍ لملء الفراغ "gap" أثناء عملية اختيار العناصر. إذ تلاحظ كيف تبني دالة `filter` مصفوفةً جديدةً من العناصر التي تجتاز اختبارها بدلًا من حذف العناصر من المصفوفة القديمة، وهذا يشير إلى أنّ هذه الدالة دالةً نقيةً pure function، إذ لا تُعدِّل المصفوفة المُمرَرة إليها.

تشبه هذه الدالة التابع `forEach` في كونها تابع مصفوفة قياسي، وقد عرَّف المثال السابق الدالة لتوضيح كيفية عملها من الداخل ليس إلا؛ أما من الآن فصاعدًا فسنستخدمها فقط كما يلي:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));
// → [{name: "Mongolian", …}, …]
```

5.6 التحويل مع map

ليكن لدينا مصفوفة كائنات تمثِّل عدة نصوص، حيث أُنتجت بترشيح مصفوفة `SCRIPTS`، لكننا نريد مصفوفةً من الأسماء لأنها أسهل في البحث والتدقيق.

هنا يأتي دور التابع `map` الذي يحوّل مصفوفةً بتطبيق دالة على جميع عناصرها، ثم يبني مصفوفةً جديدةً من القيم المعادة، وتكون المصفوفة الجديدة بطول المصفوفة المدخلة، مع إعادة توجيه محتوياتها في شكل جديد بواسطة الدالة.

```
function map(array, transform) {
  let mapped = [];
  for (let element of array) {
    mapped.push(transform(element));
  }
  return mapped;
}
```

```
let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", …]
```

وبالمثل، يُعَدّ التابع `map` تابع مصفوفة قياسي، أي يحاكي كلًا من: `filter`، و `forEach`.

5.7 التلخيص باستخدام reduce

يُعَدّ إجراء حسابات على قيمة واحدة من المصفوفات من العمليات الشائعة على هذه المصفوفات، ومثالنا التعاودي الذي يستدعي تجميعة من الأعداد هو مثال على هذا، وكذلك إيجاد النص الحاوي على أكبر عدد من المحارف.

تُدعى العملية العليا الممثِلة لهذا النمط بـ reduce، وتُدعى fold أحيانًا، إذ تُنتِج قيمةً بتكرار أخذ عنصر ما من المصفوفة، ومن ثم جمعه مع القيمة الحالية، فتبدأ عند إيجاد مجموع الأعداد من الصفر، وتضيف كل عنصر إلى المجموع الإجمالي.

تأخذ `reduce` جزءًا من المصفوفة، ودالة جامعة combining function، وقيمة بدء start value، على أساس معامِلات، وتختلف هذه الدالة عن دالتي `filter`، و `map` المتّسمتين بالوضوح والمباشرة أكثر، كما في الدالة التالية:

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

يملك تابع المصفوفة القياسي `reduce` الموافق لهذه الدالة خاصيةً مميزة، إذ يسمح لك بإهمال وسيط `start` إن كان في مصفوفتك عنصرًا واحدًا على الأقل، حيث سيأخذ العنصر الأول من المصفوفة على أساس قيمة بدء له، ويبدأ التقليل من العنصر الثاني، كما في المثال التالي:

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

يمكننا استخدام reduce مرتين لحساب عدد محارف أو كلمات نص ما، كما في المثال التالي:

```
function characterCount(script) {
  return script.ranges.reduce((count, [from, to]) => {
    return count + (to - from);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return characterCount(a) < characterCount(b) ? b : a;
}));
// → {name: "Han", …}
```

تقلل دالة characterCount المجالات المسندَة إلى نص ما بجمع أحجامها، حيث تلاحظ استخدام التفكيك في قائمة المعامِلات للدالة المقلِّلة، ثم يُستَدعى التابع reduce مرةً ثانية لإيجاد أكبر نص من خلال موازنة نصين في كل مرة وإعادة الأكبر بينهما.

تحتوي لغات الهان -نظام الكتابة الصيني، والياباني، والكوري- على أكثر من 89 ألف محرف مسنَد إليها في معيار يونيكود، مما يجعلها أكبر نظام كتابة في مجموعة البيانات.

وقد قرر مجمع الترميز الموحد Unicode Consortium معاملة تلك اللغات على أنها نظام كتابة واحد لتوفير رموز المحارف، رغم مضايقة هذا لبعض العامة، وسُمي ذلك القرار بتوحيد الهان Han Unification.

5.8 قابلية التركيب

لن يبدو مثال إيجاد أكبر نص سيئًا إذا كتبناه دون استخدام الدوال العليا فيه، كما في المثال التالي:

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", …}
```

لا يزال البرنامج سهل القراءة على الرغم من استخدام أربع رابطات جديدة، وزيادة أربعة أسطر أخرى، حيث تبرز الدوال العليا عند الحاجة لإجراء عمليات تركيب، فمثلًا، دعنا نكتب شيفرة للبحث عن السنة المتوسطة لإنشاء لغة ما سواءً كانت حيةً أو ميتةً في مجموعة البيانات:

```
function average(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1165
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 204
```

نتبين مما سبق أنَّ متوسط اللغات الميتة في اليونيكود أقدم من الحية، وهذا متوقع لا شك، كما أنَّ الشيفرة السابقة ليست صعبة القراءة، إذ يمكن النظر إليها على أنها أنبوب، حيث نبدأ فيها بجميع اللغات، ثم نرشّح الحية منها أو الميتة، وبعدها نأخذ أعوام هؤلاء ونحسب المتوسط، ثم نقرب النتيجة لأقرب رقم صحيح. كما نستطيع كتابة هذه العملية الحسابية على صورة حلقة تكرارية واحدة كبيرة، كما يلي:

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1165
```

لكن من الصعب قراءة هذا الأسلوب لمعرفة ما الذي يُحسب فيه، وبما أنّ النتائج البينية غير ممثلة على أساس قيم مترابطة، فستدور حول نفسك لاستخراج شيء مثل `average` إلى دالة منفصلة.

يختلف هذان المنظوران من حيث ما يفعله الحاسوب، إذ يبني الأول مصفوفات جديدة عند تشغيل `filter`، و`map`؛ بينما يحسب الثاني بعض الأعداد فقط وينجز عملًا أقل، ولا شك في تفضيلك للشيفرة المقروءة، لكن إن كنت تعالج مصفوفات ضخمة بتواتر، فيكون الأسلوب الأقل تجريدًا هنا أفضل بسبب السرعة الزائدة.

5.9 السلاسل النصية ورموز المحارف

تُعَدّ معرفة اللغة التي يستخدمها نص ما، إحدى استخدامات مجموعات البيانات، حيث يمَكّننا رمز المحرف المعطى من كتابة دالة لإيجاد اللغة الموافقة لهذا المحرف إن وجِدت، وذلك بسبب ارتباط كل لغة بمصفوفة من مجالات رموز المحارف، أي كما في المثال التالي:

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    })) {
      return script;
    }
  }
  return null;
}

console.log(characterScript(121));
// → {name: "Latin", …}
```

يُعَدّ تابع `some` أعلاه دالةً عليا، إذ تأخذ دالة اختبار لتخبرك إن كانت الدالة تعيد `true` لأيّ عنصر في المصفوفة، ولكن كيف سنحصل على رموز المحارف في سلسلة نصية؟

ذكرنا في الفصل الأول أنّ سلاسل جافاسكربت النصية مرمّزة على أساس تسلسلات من أعداد 16-بت، وتسمى هذه الأعداد بالأعداد البِتّية لمحارف السلسلة code units، حيث صُمِّمت رموز المحارف character code في اليونيكود لتتوافق مع وحدة unit -مثل التي تعطيك 65 ألف محرف-؛ ولكن عارض بعض العامة زيادة الذاكرة المخصصة لكل محرف بعدما تبين عدم كفاية هذا، فقد ابتُكِرت لمعالجة هذه المشكلة صيغة UTF-16 التي استخدمتها جافاسكربت، حيث تصف أكثر المحارف شيوعًا باستخدام عدد بِتّي لمحرف 16-بت واحد، لكنها تستخدم زوجًا من تلك الأعداد البِتّية لغيرها.

تُعَدّ UTF-16 فكرةً سيئةً حاليًا، إذ يبدو أنّها اختُرعت لخلق أخطاء! فمن السهل كتابة برامج تدّعي أنّ الأعداد البِتّية للمحارف والمحارف هما الشيء نفسه، وإن لم تكن لغتك تستخدم محارف من وحدتين فلا بأس؛ لكن سينهار البرنامج عند محاولة أحدهم استخدامه مع المحارف الصينية الأقل شيوعًا، ولحسن الحظ، فقد بدأ الجميع مع اختراع الإيموجي (الرموز التعبيرية) باستخدام المحارف ذات الوحدتين.

لكن العمليات الواضحة في سلاسل جافاسكربت النصية، مثل: الحصول على طولها باستخدام خاصية `length`، والوصول إلى محتواها باستخدام الأقواس المربعة، لا تتعامل إلا مع الأعداد البِتية للمحارف، أنظر إلى ما يلي:

```
// محرفي إيموجي، حصان وحذاء
let horseShoe = "🐴👟";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Invalid half-character)
console.log(horseShoe.charCodeAt(0));
// → 55357 (رمز لنصف محرف)
console.log(horseShoe.codePointAt(0));
// → 128052 (الرمز الحقيقي لرمز الحصان)
```

يعطينا تابع `charCodeAt` عداد بِتي للمحرف فقط وليس الرمز الكامل للمحرف؛ أما تابع `codePointAt` الذي أضيف لاحقًا فيعطي محرف يونيكود كامل، لذا نستطيع استخدامه للحصول على المحارف من سلسلة نصية، لكن لا يزال الوسيط الممرر إلى `codePointAt` فهرسًا داخل تسلسل من الأعداد البِتّية لمحارف السلسلة، لذا فلا زلنا في حاجة إلى النظر هل يأخذ المحرف وحدتين رمزيتين أم وحدةً واحدةً للمرور على جميع المحارف في سلسلة نصية ما.

ذكرنا في الفصل السابق أنه يمكن استخدام حلقة `for/of` التكرارية على السلاسل النصية، وقد أُدخل هذا النوع من الحلقات -شأنه في هذا شأن `codePointAt`- في الوقت الذي كانت العامة فيه على علم بمشكلة UTF-16، لذا سيعطيك محارف حقيقية حين استخدامه للتكرار على سلسلة نصية، بدلًا من أعداد بِتية لمحارف السلسلة.

```
let roseDragon = "🌹🐉";
for (let char of roseDragon) {
  console.log(char);
}
// → 🌹
// → 🐉
```

وإن كان لديك محرف -وما هو إلا سلسلة نصية من وحدة رمزية أو اثنتين-، فستستطيع استخدام `codePointAt(0)` للحصول على رمزه.

5.10 التعرف على النصوص

لـدينا دالـة characterScript، وطريقـةً للتكـرار الصـحيح على المحـارف، فـالخطوة التاليـة إذًا هي عـدّ المحارف المنتمية لكل لغة، كما في التجريد أدناه للعد:

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.findIndex(c => c.name == name);
    if (known == -1) {
      counts.push({name, count: 1});
    } else {
      counts[known].count++;
    }
  }
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

تتوقع دالة countBy تجميعة collection -من أي شيء نستطيع التكـرار عليـه باسـتخدام for/of، ودالـةً لحسـاب اسـم المجموعـة group للعنصـر المعطى، حيث تعيـد مصـفوفةً من الكائنـات وكـل منهـا هـو اسـم لمجموعة، وتخبرك بعدد العناصر الموجودة في تلك المجموعة.

تستخدِم هذه الدالة تابع مصفوفة اسمه findIndex، حيث يحـاكي indexof نوعًـا مـا، لكنـه يبحث في القيمة الأولى التي تعيد true في الدالة المعطاة بدلًا من البحث عن قيمة معينة، كما يتشابه معه في إعادة -1 عند عدم وجود مثل هذا العنصر، ونستطيع باستخدام countBy كتابة الدالة التي تخبرنـا أيّ اللغـات مسـتخدمة في نص ما.

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "none";
  }).filter(({name}) => name != "none");
```

```
  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No scripts found";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

تَعُدّ الدالة أولًا المحارف من خلال الاسم باستخدام `characterScript` لتعيينها اسمًا وتعود إلى السلسلة `"none"` من أجل المحارف التي ليست جزءًا من أي لغة، ثم يحذف استدعاء `filter` الإدخال الخاص بـ `"none"` من المصفوفة الناتجة بما أننا لا نهتم بتلك المحارف.

سنحتاج إلى العدد الإجمالي للمحارف المنتمية إلى لغة ما لنستطيع حساب النسب، ويمكن معرفة ذلك من خلال `reduce`، وإن لم نجد هذه المحارف، فستعيد الدالة سلسلةً نصيةً محدّدةً، وإلا فستحوّل مدخلات العد إلى سلاسل نصية مقروءة باستخدام `map`، ومن ثم تدمجها باستخدام `join`.

5.11 خاتمة

تبين لنا مما سبق أنّ تمرير قيم دالة ما إلى دوال أخرى مفيد جدًا، إذ يسمح لنا بكتابة دوال تُنمذِج الحسابات التي بها فراغات، إذ تستطيع الشيفرة التي تستدعي هذه الدوال ملء تلك الفراغات بتوفير قيم الدوال؛ أما المصفوفات فتعطينا عددًا من التوابع العليا، ويمكننا استخدام `forEach` للتكرار على عناصر داخل مصفوفة ما؛ ويعيد تابع `filter` مصفوفةً جديدةً تحتوي العناصر التي تمرِّر دالة التوقّع predicate function؛ كما نستطيع تحويل مصفوفة ما من خلال وضع كل عنصر في دالة باستخدام `map`؛ وكذلك نستطيع استخدام `reduce` لجمع عناصر مصفوفة ما داخل قيمة واحدة؛ أما تابع `some` فينظر هل ثَمّ عنصر مطابق لدالة توقع معطاة أم لا؛ ويبحث `findIndex` عن موضع أول عنصر مطابق لتوقّع ما.

5.12 تدريبات

5.12.1 التبسيط

استخدم تابع `method`، و`concat` لتبسيط مصفوفة بها مصفوفات أخرى، إلى مصفوفة واحدة بها جميع العناصر الموجودة في تلك المصفوفات كلها.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
let arrays = [[1, 2, 3], [4, 5], [6]];
// ضع شيفرتك هنا.
// → [1, 2, 3, 4, 5, 6]
```

5.12.2 الحلقة التكرارية الخاصة بك

اكتب دالة `loop` العليا التي تعطي شيئًا مثل تعليمة حلقة `for` التكرارية، إذ تأخذ قيمةً، ودالة اختبار، ودالة تحديث، ومتن دالة.

تستخدِم في كل تكرار دالة الاختبار أولًا على قيمة التكرار الحالية، وتتوقف إن لم تتحقق -أي أعادت false-، ثم تستدعي متن الدالة لتعطيه القيمة الحالية، وأخيرًا تستدعي دالة التحديث لإنشاء قيمة جديدة والبدء من جديد. تستطيع عند تعريف الدالة استخدام حلقة تكرارية عادية لتنفيذ التكرار الفعلي.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// شيفرتك هنا.

loop(3, n => n > 0, n => n - 1, console.log);
// → 3
// → 2
// → 1
```

5.12.3 كل شيء

تحتوي المصفوفات على تابع `every` بالتماثل مع تابع `some`، ويتحقق `every` إذا تحققت الدالة المعطاة لكل عنصر في المصفوفة، ويمكن النظر إلى `some` في سلوكه على المصفوفات على أنه عامِل `||`، في حين يكون `every` عامِل `&&`.

استخدم `every` على أساس دالة تأخذ مصفوفة ودالة توقّع على أساس معامِلات، واكتب نسختين، إحداهما باستخدام حلقة تكرارية، والأخرى باستخدام تابع `some`.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
function every(array, test) {
  // ضع شيفرتك هنا.
}

console.log(every([1, 3, 5], n => n < 10));
// → true
console.log(every([2, 4, 16], n => n < 10));
// → false
console.log(every([], n => n < 10));
// → true
```

إرشادات الحل

يستطيع التابع `every` إيقاف تقييم المزيد من العناصر بمجرد إيجاد عنصر واحد غير مطابق، تمامًا كما في حالة عامل `&&`، لذا تستطيع النسخة المبنية على الحلقة التكرارية القفز خارجها -باستخدام `break`، أو `return`- عند إيجاد العنصر الذي تعيد له دالة التوقّع false، فإذا انتهت الحلقة التكرارية دون مقابلة عنصر كهذا، فسنعرف بتطابق جميع العناصر ويجب لإعادة `true`.

نستخدم قوانين دي مورجَن De Morgan لبناء `every` فوق `some`، والذي ينص على أنّ `a && b` تساوي `!(!a || !b)`، ويمكن أن يُعمَّم هذا للمصفوفات، حيث تكون كل العناصر في المصفوفة مطابقةً إذا لم يكن في المصفوفة عنصرًا غير مطابق.

5.12.4 اتجاه الكتابة السائد

اكتب دالة تحسب اتجاه الكتابة السائد في نص ما، وتذكّر أنّه لدى كل كائن من كائنات اللغات خاصية `direction`، والتي من الممكن أن تكون: ltr، أو rtl، أو ttb، كما ذكرنا في سابق شرحنا هنا.

الاتجاه السائد هو اتجاه أغلب المحارف المرتبطة بلغة ما، وستستفيد من دالتي: `characterScript` و`countBy` المعرَّفتَين في هذا الفصل.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
function dominantDirection(text) {
  // ضع شيفرتك هنا.
}

console.log(dominantDirection("Hello!"));
```

```
// → ltr
console.log(dominantDirection("Hey, مساء الخير"));
// → rtl
```

إرشادات الحل

قد يبدو حلك مثل النصف الأول من مثال `textScripts`، فعليك عدّ المحارف بمقياس مبني على `characterScript`، ثم ترشيح الجزء الذي يشير إلى المحارف غير المهمة (غير النصية).

يمكن إيجاد الاتجاه الذي يحمل أعلى عدد من المحارف بواسطة `reduce`، فإذا لم يكن ذلك واضحًا فارجع إلى المثال السابق في هذا الفصل حيث استُخدِم `reduce` لإيجاد النص الذي فيه أكثر المحارف.

دورة تطوير تطبيقات الويب
باستخدام لغة Ruby

دورة تدريبية متكاملة من الصفر وحتى الاحتراف
تمكنك من التخصص في هندسة الويب ودخول سوق العمل

التحق بالدورة الآن

6. الحياة السرية للكائنات

يلاحَظ نوع البيانات المجرد بكتابة برنامج خاص يعرِّف النوع من حيث العمليات التي يمكن تنفيذها عليه.

— باربرا ليزكوف Barbara Liskov، البرمجة بالأنواع المجردة للبيانات.

تحدثنا في الفصل الرابع عن الكائنات في جافاسكربت، ولدينا في ثقافة البرمجة شيء يسمى بالبرمجة كائنية التوجه، وهي مجموعة تقنيات تستخدم الكائنات والمفاهيم المرتبطة بها مثل مبدأ مركزي لتنظيم البرامج. ورغم عدم وجود إجماع على التعريف الدقيق للبرمجة كائنية التوجه هذه، إلا أنها قد غيرت شكل لغات برمجة كثيرة من حيث تصميمها، بما فيها جافاسكربت، وسنتعرض في هذا الفصل للطرق التي يمكن تطبيق أفكار هذا المفهوم في جافاسكربت.

6.1 التغليف Encapsulation

تتلخص فكرة البرمجة كائنية التوجه في تقسيم البرامج إلى أجزاء صغيرة وجعل كل جزء مسؤولًا عن إدارة حالته الخاصة، وهكذا يمكن حفظ المعلومات الخاصة بالأسلوب الذي يعمل به جزء ما من البرنامج داخل ذلك الجزء فقط محليًا، بحيث إذا عمل شخص ما على جزء آخر من البرنامج، فليس عليه معرفة أو إدراك حتى هذه البيانات والمعلومات؛ وإذا تغيرت تلك التفاصيل المحلية، فلن نحتاج سوى إلى تعديل جزء الشيفرة المتعلق بها فقط. ويُطلق على فصل الواجهة عن الاستخدام نفسه أو التطبيق بالتغليف، وهو فكرة عظيمة.

تتفاعل الأجزاء المختلفة من البرامج مع بعضها البعض من خلال واجهات interfaces، وهي مجموعات محدودة من الدوال أو الرابطات bindings التي توفر أداءً مفيدًا في المستويات العليا التجريدية التي تخفي استخدامها الدقيق والمباشر. كما نُمذِجت مثل تلك الأجزاء باستخدام كائنات، وواجهاتها مكونة من مجموعات

محددة من التوابع والخصائص، حيث يوجد نوعان من هذه الخصائص، إما عامة عندما تكون جـزءًا من الواجهـة، أو خاصة يجب ألا يقربها أي شيء خارج الشيفرة.

توفر الكثير من اللغات طريقةً للتمييز بين الخصائص العامة والخاصة، وذلك لمنع الشـيفرات الخارجيـة من الوصول إلى الخصائص الخاصة؛ أما جافاسكربت فلا تفعل ما سبق، إذ تتبـع أسـلوبها البسـيط في ذلـك حاليًـا، ويوجد ثمة أعمال لإضافة ذلك إليها. رغم عدم دعم اللغة لهذه الخاصية في التفرقة، إلا أنّ مـبرمجي جافاسـكربت يفعلون ذلك من حيث المبدأ، فالواجهـة المتاحـة موصـوفة ومشـروحة في التوثيـق أو التعليقـات، ومن الشـائع كذلك وضع شرطة سفلية (_) في بداية أسماء الخصائص للإشارة إلى أنها "خاصة".

6.2 التوابع Methods

التوابع ليست إلا خصائص حاملة لقيم الدوال، انظر المثال التالي لتابع بسيط:

```
let rabbit = {};
rabbit.speak = function(line) {
  console.log(`The rabbit says '${line}'`);
};

rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

يُتوقع من التابع فعل شيء بالكائن الذي استدعي له، فحين تُستدعى دالة على أساس تابع -يُبحث عنها على أساس خاصية، ثم تُستدعى مباشرةً كما في حالة `object.method()`-، ستشير الرابطة التي اسـتدعت `this` في متنها مباشرةً إلى الكائن الذي استُدعي عليه.

```
function speak(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit = {type: "white", speak};
let hungryRabbit = {type: "hungry", speak};

whiteRabbit.speak("Oh my ears and whiskers, " +
                  "how late it's getting!");
// → The white rabbit says 'Oh my ears and whiskers, how
//   late it's getting!'
hungryRabbit.speak("I could use a carrot right now.");
// → The hungry rabbit says 'I could use a carrot right now.'
```

فكر في this على أساس معامِل إضافي يُمرَّر في صورة مختلف، فإذا أردت تمريره صراحةً، فستستخدِم تابع call الخاص بالدالة والذي يأخذ قيمة this على أساس وسيطها الأول، وتعامِل الوسائط التالية على أساس معامِلات عادية.

```
speak.call(hungryRabbit, "Burp!");
// → The hungry rabbit says 'Burp!'
```

وبما أنّ كل دالة لها رابطة this الخاصة بها، والتي تعتمد قيمتها على الطريقة التي المَستدعاة بها، فلا تستطيع الإشارة إلى this بنطاق مغلِّف في دالة عادية معرَّفة بكلمة function المفتاحية؛ أما الدوال السهمية فتختلف في عدم ارتباط this الخاص بها، لكنها تستطيع رؤية رابطة this للنطاق الذي حولها، وعليه ستستطيع تنفيذ شيء مثل ما في الشيفرة التالية، حيث تشير إلى this مرجعيًا من داخل دالة محلية:

```
function normalize() {
  console.log(this.coords.map(n => n / this.length));
}
normalize.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

فلو كتبنا الوسيط إلى map باستخدام كلمة function المفتاحية، فلن تعمل الشيفرة.

6.3 النماذج الأولية Prototypes

انظر المثال التالي:

```
let empty = {};
console.log(empty.toString);
// → function toString(){…}
console.log(empty.toString());
// → [object Object]
```

أرأيت كيف سحبنا خاصيةً من كائن فارغ!؟ حيث لم نزد على الطريقة التي تعمل بها كائنات جافاسكربت في حفظ البيانات الخاصة، لأن أغلب الكائنات لها نماذج أولية إضافةً إلى مجموعة خصائصها، وتلك النماذج الأولية ما هي إلا كائنات أخرى مستخدَمة على أساس مصدر احتياطي fallback للخصائص، فإذا طُلِب من كائن خاصية لا يملكها، فسيُبحث في نموذجه الأولي عن تلك الخاصية، ثم النموذج الأولي لذلك النموذج، وهكذا.

طيب، ما النموذج الأولي لذاك الكائن الفارغ؟ إنه object.prototype الذي يسبق الكائنات كلها، فإذا قلنا أنّ علاقات النماذج الأولية في جافاسكربت تكوِّن هيكلًا شجريًا، فسيكون جذر تلك الشجرة هو

`Object.prototype`، إذ يوفِّر بعضَ التوابع التي تظهر في جميع الكائنات الأخرى مثل `toString` الذي يحول الكائن إلى تمثيل نصي string representation.

لا تملك العديد من الكائنات `Object.prototype` مثل نموذجها الأولي، بل يكون لها كائنٌ آخر يوفر مجموعةً مختلفةً من الخصائص الافتراضية.

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

كما تتوقع من المثال السابق، سيُعيد `Object.getPrototypeOf` النموذج الأولي للكائن.

تنحدر الدوال من `Function.prototype`؛ أما المصفوفات فتنحدر من `Array.prototype`، كما في المثال التالي:

```
console.log(Object.getPrototypeOf(Math.max) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
            Array.prototype);
// → true
```

سيكون لكائن النموذج الأولي المشابه لهذا، نموذج أولي خاص به وهو `Object.prototype` غالبًا، وذلك لاستمراره بتوفير توابع مثل `toString`؛ وتستطيع استخدام `Object.create` لإنشاء كائن مع نموذج أولي بعينه، كما في المثال التالي:

```
let protoRabbit = {
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
};
let killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!'
```

تُعَدّ خاصية مثل speak(line) في تعبير الكائن طريقةً مختصرةً لتعريف تابع ما، إذ تنشِئ خاصيةً اسمها speak، وتعطيها دالةً على أساس قيمة لها؛ كما يتصرف الأرنب "proto" في المثال السابق على أساس حاوية للخصائص التي تشترك فيها جميع الأرانب؛ أما في حالة مثل الأرنب القاتل killer rabbit، فيحتوي على خصائص لا تنطبق إلا عليه -نوعه في هذه الحالة-، كما يأخذ خصائص مشتركة من نموذجه الأولي.

6.4 الأصناف Classes

يحاكي نظام النماذج الأولية في جافاسكربت مفهوم الأصناف Classes في البرمجة كائنية التوجه، حيث تحدِّد هذه الأصناف الشكل الذي سيكون عليه نوع ما من كائن، وذلك بتحديد توابعه وخصائصه، كما يُدعى مثل ذلك الكائن بنسخة instance من الصنف.

تُعَدّ النماذج الأولية مفيدةً هنا في تحديد الخصائص المشتركة بين جميع نُسَخ الصنف التي لها القيمة نفسها مثل التوابع؛ أما الخصائص المختلفة بين كل نسخة -كما في حالة خاصية type لأرنبنا في المثال السابق-، فيجب تخزينها في الكائن نفسه مباشرةً. لذا عليك إنشاء كائنًا مشتقًا من النموذج الأولي المناسب من أجل إنشاء نسخة من صنف ما، لكن في الوقت نفسه يجب التأكد من امتلاكه الخصائص الواجب وجودها في نُسَخ ذلك الصنف، وهذا ما يمثل وظيفة دالة الباني constructor، انظر ما يلي:

```
function makeRabbit(type) {
  let rabbit = Object.create(protoRabbit);
  rabbit.type = type;
  return rabbit;
}
```

توفر جافاسكربت طريقةً لتسهيل تعريف هذا النوع من الدوال، فإذا وضعتَ كلمة new المفتاحية أمام استدعاء الدالة مباشرةً، فستُعامَل الدالة على أساس باني، وهذا يعني أنه سيُنشَأ الكائن الذي يحمل النموذج الأولي المناسب تلقائيًا، بحيث يكون مقيدًا بـ this في الدالة، ثم يُعاد في نهاية الدالة، ويمكن العثور على كائن النموذج الأولي المستخدَم عند بناء الكائنات من خلال أخذ خاصية protoype لدالة الباني.

```
function Rabbit(type) {
  this.type = type;
}
Rabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
};

let weirdRabbit = new Rabbit("weird");
```

تحصل البواني، بل كل الدوال، على خاصية اسمها prototype تحمل بدورها كائنًا فارغًا مشتقًا من Object.prototype، وتستطيع استبدال كائن جديد به إن شئت أو إضافة خصائص إلى الكائن الجديد كما في المثال.

تتكوّن أسماء البواني من الحروف الكبيرة لتمييزها عما سواها، ومن المهم إدراك الفرق بين الطريقة التي يرتبط بها النموذج الأولي بالباني من خلال خاصية prototype، والطريقة التي يكون للكائنات فيها نماذج أولية -والتي يمكن إيجادها باستخدام Object.getPrototypeOf-.

Function.Prototype هو النموذج الأولي الفعلي للباني بما أنّ البواني ما هي إلا دوال في الأصل، وتحمل خاصية prototype الخاصة به النموذج الأولي المستخدَم للنسخ التي أنشِئت من خلاله.

```
console.log(Object.getPrototypeOf(Rabbit) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(weirdRabbit) ==
            Rabbit.prototype);
// → true
```

6.5 صياغة الصنف Class Notation

ذكرنا أن أصناف جافاسكربت ما هي إلا دوال بانية مع خاصية النموذج الأولي، وقد كان ذلك حتى عام 2015؛ أما الآن فقد تحسنت الصيغة التي صارت عليها كثيرًا، انظر إلى ما يلي:

```
class Rabbit {
  constructor(type) {
    this.type = type;
  }
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
}

let killerRabbit = new Rabbit("killer");
let blackRabbit = new Rabbit("black");
```

تبدأ كلمة class المفتاحية تصريح صنفٍ يسمح لنا بتعريف باني ومجموعة توابع في مكان واحد، كما يمكن كتابة أيّ عدد من التوابع بين قوسي التصريح، لكن يُعامَل التابع الحامل لاسم constructor معاملةً خاصةً، إذ يوفِّر وظيفة الباني الفعلية التي ستكون مقيدة بالاسم Rabbit، في حين تُحزَّم التوابع الأخرى في

النموذج الأولي لذلك الباني، ومن ثم يكون تصريح الصنف الذي ذكرناه قبل قليل مكافئًا لتعريف الباني من القسم السابق، كونه يبدو أفضل للقارئ.

ولا تسمح تصريحات الأصناف حاليًا إلا بإضافة التوابع إلى النموذج الأولي، وهي الخصائص التي تحمل دوالًا، رغم أن ذلك قد يكون مرهقًا إذا أردت حفظ قيمة غير دالّية non-function هناك، وقد يتحسن ذلك في الإصدار القادم من اللغة، لكن حتى ذلك الحين تستطيع إنشاء مثل تلك الخصائص بتغيير النموذج الأولي مباشرةً بعد تعريف الصنف.

يمكن استخدام `class` في التعليمات والتعابير على حد سواء، وشأنها في ذلك شأن `function`، حيث لا تعرِّف رابطةً عند استخدامها على أساس تعبير، وإنما تنتج الباني كقيمة فقط. وتستطيع إهمال اسم الصنف في تعبير الصنف، كما في المثال التالي:

```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

6.6 إعادة تعريف الخصائص المشتقة

تُضاف الخاصية إلى الكائن نفسه عند إضافتها إليه سواءً كان موجودًا في النموذج الأولي أم غير موجود، فإن كان ثمة خاصية موجودة بالاسم نفسه في النموذج الأولي، فلن تؤثِّر هذه الخاصية في الكائن بما أنها مخفية الآن خلف الخاصية التي يملكها الكائن.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

يبيّن المخطط التالي الموقف بعد تشغيل الشيفرة السابقة، إذ يقبع النموذجين الأوليَين لـ `Rabbit`، و`Object` خلف `killerRabbit` على أساس حاجز خلفي له، بينما يُبحث عن الخصائص التي ليست موجودة في الكائن.

وتبدو فائدة إعادة تعريف الخصائص properties overriding الموجودة في النموذج الأولي في التعبير عن الخصائص الاستثنائية في نُسَخ الأصناف العامة للكائنات، كما في مثال أسنان الأرنب rabbit teeth السابق، مع السماح للكائنات غير الاستثنائية بأخذ قيمة قياسية من نموذجها الأولي.

كما يمكن استخدام إعادة التعريف لإعطاء تابع `toString` للنماذج الأولية للدالة والمصفوفة القياسيتين، بحيث يختلف عن النموذج الأساسي للكائن.

```
console.log(Array.prototype.toString ==
            Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

يعطي استدعاء `toString` على مصفوفة نتيجةً محاكيةً لاستدعاء `.join(",")` عليها، إذ تضع فواصل إنجليزية بين القيم الموجودة في المصفوفة؛ أما الاستدعاء المباشر لـ `Object.prototype.toString` مع مصفوفة، فينتج سلسلةً نصيةً مختلفةً، حيث تضع كلمة object واسم النوع بين أقواس مربعة، وذلك لعدم معرفة تلك الدالة بشأن المصفوفات، كما في المثال التالي:

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

6.7 الخرائط Maps

استخدمنا كلمة map في الفصل السابق في عملية تحويل هيكل البيانات بتطبيق دالة على عناصره، رغم بعد معنى الكلمة نفسها، التحويل، الدال عن الفعل الذي تنفذه، وهنا أيضًا وفي البرمجة عمومًا، فتُستخدَم هذه الكلمة كذلك لغرض مختلف لكنه قريب مما رأينا، وكلمة map على أساس اسم هي أحد أنواع هياكل البيانات

الذي يربط القيم (المفاتيح) بقيم أخرى، فإذا أردت ربط الأسماء بأعمار مقابلة لها، فتستطيع استخدام كائنات لذلك، كما في المثال التالي:

```
let ages = {
  Ziad: 39,
  Hasan: 22,
  Sumaia: 62
};

console.log(`Sumaia is ${ages["Sumaia"]}`);
// → Sumaia is 62
console.log("Is Jack's age known?", "Jack" in ages);
// → Is Jack's age known? false
console.log("Is toString's age known?", "toString" in ages);
// → Is toString's age known? true
```

أسماء خصائص الكائن هنا هي أسماء الناس المذكورة في المثال، وقيم الخصائص هي أعمارهم، لكننا بالتأكيد لم نذكر أيّ شخص اسمه toString في تلك الرابطة، لكن لأن الكائنات العادية مشتقة من `Object.prototype` فيبدو الأمر وكأن الخاصية موجودة هناك، لهذا فمن الخطر معاملة الكائنات العادية مثل معاملة خرائط -النوع Map- هنا.

لدينا عدة طرق مختلفة لتجنب هذه المشكلة، فمن الممكن مثلًا إنشاء كائنات بدون نموذج أولي، حتى إذا مرّرت `null` إلى `Object.create`، فلن يكون الكائن الناتج مشتقًا من `Object.prototype`، وعليه يمكن استخدامه بأمان على أساس خارطة.

```
console.log("toString" in Object.create(null));
// → false
```

يجب أن تكون أسماء خصائص الكائنات سلاسل نصية، فإن أردت ربطًا لا يمكن تحويل مفاتيحه بسهولة إلى سلاسل نصية -مثل الكائنات- فلا تستخدم كائنًا على أساس خارطة، ولحسن الحظ فتملك جافاسكربت صنفًا اسمه `Map` مكتوب لهذا الغرض خاصة، حيث يخزِّن حالة الربط ويسمح بأي نوع من المفاتيح.

```
let ages = new Map();
ages.set("Ziad", 39);
ages.set("Hasan", 22);
ages.set("Sumaia", 62);
```

```
console.log(`Sumaia is ${ages.get("Sumaia")}`);
// → Sumaia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
// → Is Jack's age known? false
console.log(ages.has("toString"));
// → false
```

تُعَدّ التوابع set، وget، وhas جزءًا من واجهة كائن Map، فليس من السهل كتابة هيكل بيانات لتحديث مجموعة كبيرة من القيم والبحث فيها، ولكن لا تقلق، فقد كفانا شخص آخر مؤنة ذلك، حيث نستطيع استخدام ما كتبه من خلال تلك الواجهة البسيطة.

إذا أردت معاملة كائن عادي لديك على أساس خارطة (النوع Map) لسبب ما، فمن المهم معرفة أن Object.keys يعيد المفاتيح الخاصة بالكائن فقط، وليس تلك الموجودة في النموذج الأولي، كما تستطيع استخدام التابع hasOwnProperty على أساس بديل لعامِل in، حيث يتجاهل النموذج الأولي للكائن، كما في المثال التالي:

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

6.8 تعددية الأشكال Polymorphism

إذا استدعيتَ دالة String -التي تحوِّل القيمة إلى سلسلة نصية- على كائن ما، فستستدعي التابع toString على ذلك الكائن لمحاولة إنشاء سلسلة نصية مفيدة منه.

كما ذكرنا سابقًا، تعرِّف بعض النماذج الأولية القياسية إصدارًا من toString خاصًا بها، وذلك لتستطيع إنشاء سلسلة نصية تحتوي بيانات مفيدة أكثر من "[object Object]"، كما تستطيع فعل ذلك بنفسك إن شئت.

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
};

console.log(String(blackRabbit));
// → a black rabbit
```

وهذه صورة بسيطة من مفهوم بالغ القوة والأثر، فإن كُتب جزء من شيفرة ما ليعمل مع كائنات بها واجهة معينة -تابع `toString` في هذه الحالة-، فيمكن إلحاق أي نوع من الكائنات الداعمة لتلك الواجهة بالشيفرة، حيث ستعمل دون مشاكل؛ وتسمى تلك التقنية بتعددية الأشكال، وتعمل الشيفرة المتعددة الأشكال مع قيم ذات أشكال مختلفة طالما أنها تدعم الواجهة التي تتوقعها.

كما ذكرنا في الفصل الرابع، تستطيع حلقة `for/of` التكرار على عدة أنواع من هياكل البيانات، وتلك حالة أخرى من تعددية الأشكال، حيث تتوقع مثل تلك الحلقات التكرارية من هيكل البيانات أن يكشف واجهة معينة، وهو ما تفعله المصفوفات والسلاسل النصية؛ كما نستطيع إضافة تلك الواجهة إلى كائناتنا الخاصة، لكننا نحتاج إلى معرفة ما هي الرموز symbols قبل فعل ذلك.

6.9 الرموز Symbols

تستطيع عدة واجهات استخدام اسم الخاصية نفسها لأشياء عدة، فمثلًا، نستطيع تعريف واجهة بحيث يحوِّل فيها التابع `toString` الكائن إلى قطعة من خيوط الغزل، لكن من غير الممكن لكائن أن يتوافق مع تلك الواجهة ومع الاستخدام القياسي لـ `toString`.

هذه المشكلة سيئة لكنها لا تشغل بال من يكتب بجافاسكربت لأنها غير شائعة، ورغم هذا فقد وفر مصممو جافاسكربت لنا حلًا لهذه المشكلة، إذ أن تلك من وظيفتهم على أي حال.

حين زعمنا أن أسماء الخصائص هي سلاسل نصية لم نكن محقين 100%، فرغم أنها حقًا سلاسل نصية إلا قد تكون رموزًا أيضًا، وهي -أي الرموز- قيم أنشِئت بواسطة دالة `Symbol`، كما تُعَدّ الرموز المنشَئة حديثًا فريدةً، على عكس السلاسل النصية، بحيث لا تستطيع إنشاء الرمز نفسه مرتين.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

تُضمَّن السلسلة النصية الممررة إلى `Symbol` تلقائيًا حين تحوّلها إلى سلسلة نصية، كما تسهِّل التعرف على الرمز عند عرضه في الطرفية console مثلًا؛ ولأن الرموز فريدة ويمكن استخدامها على أساس أسماء للخصائص، فهي مناسبة لتعريف الواجهات التي يمكن وجودها مع الخصائص الأخرى مهما كانت أسماؤها.

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
  return `${this.length} cm of blue yarn`;
```

```
};

console.log([1, 2].toString());
// → 1,2
console.log([1, 2][toStringSymbol]());
// → 2 cm of blue yarn
```

ومن الممكن إضافة خصائص رمز ما في الأصناف وتعبيرات الكائنات باستخدام أقواس مربعة حول اسم الخاصية، وبسبب ذلك سيُقيَّم اسم الخاصية مثل صيغة الوصول إلى الخاصية التي تستخدِم قوسين مربعين، حيث سيسمح لنا هذا بالإشارة إلى الرابطة التي تحمل الرمز.

```
let stringObject = {
  [toStringSymbol]() { return "a jute rope"; }
};
console.log(stringObject[toStringSymbol]());
// → a jute rope
```

6.10 واجهة المكرر iterator

يُتوقع من الكائن المعطى لحلقة `for/of` قابليته للتكرار، ويعني ذلك أنّ به تابعًا مسمى مع الرمز `Symbol.iterator`، وهو قيمة رمز معرَّفة من قِبَل اللغة، ومخزَّنة على أساس خاصية لدالة `Symbol`، كما يجب على ذلك التابع إعادة كائن يوفر واجهةً ثانية تكون هي المكرِّر iterator الذي يقوم بعملية التكرار، ولديه تابع `next` الذي يعيد النتيجة التالية التي يجب أن تكون بدورها كائنًا مع خاصية `value` التي توفر القيمة التالية إن كانت موجودة، وخاصية `done` التي تعيد true إن لم تكن ثمة نتائج أخرى، وتعيد false إن كان ثَمَّ نتائج بعد.

لاحظ أن أسماء الخصائص: `next`، و`value`، و`done`، هي سلاسل نصية عادية وليست رموزًا؛ أما الرمز الوحيد هنا فهو `Symbol.iterator`، والذي سيضاف غالبًا إلى كائنات كثيرة، كما نستطيع استخدام تلك الواجهة بأنفسنا كما يلي:

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

دعنا نطبق هنا هيكل بيانات قابلًا للتكرار، حيث سنبني صنفَ matrix يتصرف على أساس مصفوفة ثنائية الأبعاد.

```
class Matrix {
  constructor(width, height, element = (x, y) => undefined) {
    this.width = width;
    this.height = height;
    this.content = [];

    for (let y = 0; y < height; y++) {
      for (let x = 0; x < width; x++) {
        this.content[y * width + x] = element(x, y);
      }
    }
  }

  get(x, y) {
    return this.content[y * this.width + x];
  }
  set(x, y, value) {
    this.content[y * this.width + x] = value;
  }
}
```

يخزِّن الصنف محتوياته في مصفوفة واحدة من عنصرين فقط، هما: العرض، والطول width×height، وتُخزَّن العناصر صفًا صفًا، فيُخزن العنصر الثالث في الصف الخامس مثلًا -باستخدام الفهرسة الصفرية التي تبدأ من الصفر- في الموضع 2 + width × 4.

تأخذ دالة الباني العرض، والطول، ودالة `element` اختيارية ستُستخدم لكتابة القيم الابتدائية؛ أما لجلب العناصر وتحديثها في المصفوفة الثنائية، فلدينا التابعان `get`، و`set`.

حين نكرر على مصفوفة ما، فنحن بحاجة إلى معرفة موضع العناصر إضافة إلى العناصر نفسها، لذا سنجعل المكرِّر ينتج كائنات لها خصائص `x`، و`y`، و`value`.

```
class MatrixIterator {
  constructor(matrix) {
    this.x = 0;
    this.y = 0;
```

```
    this.matrix = matrix;
  }

  next() {
    if (this.y == this.matrix.height) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matrix.get(this.x, this.y)};
    this.x++;
    if (this.x == this.matrix.width) {
      this.x = 0;
      this.y++;
    }
    return {value, done: false};
  }
}
```

يتتبع الصنف سير التكرار على المصفوفة الثنائية في الخصائص `x`، و`y`، ويبدأ التابع `next` بالتحقق من الوصول لأسفل المصفوفة الثنائية، فإن لم يصل إليه، فسينشِئ الكائن الذي يحمل القيمة الحالية أولًا، ثم يحدِّث موضعه، وبعد ذلك ينقله إلى السطر التالي إن تطلب الأمر.

دعنا نهيئ صنف `Matrix` ليكون قابلًا للتكرار، وانتبه إلى استخدامنا المعالجة اللاحقة للنموذج الأولي بين الحين والآخر في هذا الكتاب لإضافة توابع إلى الأصناف، وذلك لتبقى الأجزاء المفردة من الشيفرة صغيرةً ومستقِلة؛ أما في البرامج العادية التي لا تحتاج فيها إلى تقسيم الشيفرة إلى أجزاء صغيرة، فستصرِّح عن هذه التوابع مباشرةً في الصنف.

```
Matrix.prototype[Symbol.iterator] = function() {
  return new MatrixIterator(this);
};
```

نستطيع الآن تطبيق التكرار على مصفوفة ما باستخدام `for/of`.

```
let matrix = new Matrix(2, 2, (x, y) => `value ${x},${y}`);
for (let {x, y, value} of matrix) {
  console.log(x, y, value);
}
```

```
// → 0 0 value 0,0
// → 1 0 value 1,0
// → 0 1 value 0,1
// → 1 1 value 1,1
```

6.11 التوابع الجالبة والضابطة والساكنة

تتكون الواجهات من التوابع غالبًا، وقد تتضمن خصائص بها قيم غير دالية، فمثلًا، تملك كائنات `Map` خاصية `size`، والتي تخبرك كم عدد المفاتيح المخزَّنة فيها.

ليس من الضروري لمثل هذا الكائن أن يحسب ويخزن خاصية مشابهة لتلك مباشرةً في النسخة instance التي لديه، بل حتى الخصائص التي يمكن الوصول إليها مباشرةً قد تخفي استدعاءً إلى تابع، حيث تسمى مثل تلك التوابع بالتوابع الجالبة getters، وتُعرَّف بكتابة `get` أمام اسم التابع في تعبير الكائن أو تصريح الصنف.

```
let varyingSize = {
  get size() {
    return Math.floor(Math.random() * 100);
  }
};

console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

يُستدعى التابع المرتبط بخاصية `size` للكائن كلما قرأ أحد من منها، وتستطيع تنفيذ شيء مشابه حين يكتب أحدهم في خاصية ما باستخدام تابع ضابط setter.

```
class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
```

```
  }

  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

يسمح لك صنف `Temperature` في المثال أعلاه بقراءة درجة الحرارة وكتابتها سواءً بمقياس السليزيوس أو الفهرنهايت، لكنها تخزِّن داخلها درجات السليزيوس فقط، وتحوِّل من وإلى سليزيوس في التابع الجالب والضابط لـ `fahrenheit` تلقائيًا.

قد تحتاج أحيانًا إلى إلحاق بعض الخصائص لدالة الباني الخاصة بك مباشرةً بدلًا من النموذج الأولي، ولا تملك مثل تلك التوابع وصولًا إلى نسخة صنف، لكن يمكن استخدامها لتوفير طرق بديلة وإضافية لإنشاء النسخ.

تُخزَّن التوابع المكتوبة قبل اسمها `static` على الباني، وذلك داخل التصريح عن الصنف، وعليه فيسمح لك صنف `Temperature` بكتابة `Temperature.fromFahrenheit(100)` لإنشاء درجة حرارة باستخدام مقياس فهرنهايت.

6.12 الوراثة Inheritance

تتميز بعض المصفوفات بأنها تماثلية symmetric، بحيث إذا عكست إحداها حول قطرها الذي يبدأ من أعلى اليسار فستبقى كما هي ولا تتغير أي ستبقى القيمة المخزنة في الموضع (x,y) كما هي في الموضع (y,x).

تخيل أننا نحتاج إلى هيكل بيانات مثل `Matrix`، لكن يجب ضمان تماثلية المصفوفة وبقائها كذلك، وهنا نستطيع كتابة هذا من الصفر، لكننا سنكرر شيفرةً مشابهةً كثيرًا لما كتبناه سابقًا.

يسمح نظام النموذج الأولي في جافاسكربت بإنشاء صنف جديد محاكي لصنف قديم لكن مع تعريفات جديدة لبعض خصائصه، ويكون النموذج الأولي للصنف الجديد مشتقًا من القديم لكن مع إضافة تعريف جديد

إلى التابع set مثلًا، ويسمى ذلك بالاكتساب أو الوراثة inheritance، إذ يرث الصنف الجديد خصائصه وسلوكه من الصنف القديم.

```
class SymmetricMatrix extends Matrix {
  constructor(size, element = (x, y) => undefined) {
    super(size, size, (x, y) => {
      if (x < y) return element(y, x);
      else return element(x, y);
    });
  }

  set(x, y, value) {
    super.set(x, y, value);
    if (x != y) {
      super.set(y, x, value);
    }
  }
}

let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);
console.log(matrix.get(2, 3));
// → 3,2
```

يشير استخدام كلمة extends إلى وجوب عدم اعتماد هذا الصنف على النموذج الأولي الافتراضي Object مباشرةً، وإنما على صنف آخر يسمى بالصنف الأب superclass؛ أما الصنف المشتق فيكون اسمه الصنف الفرعي، أو الابن subclass.

يستدعي الباني لتهيئة نسخة من SymmetricMatrix باني صنف الأب من خلال كلمة super المفتاحية، وهذا ضروري لأنّ الكائن الجديد سيحتاج إلى خصائص النسخة التي تملكها المصفوفات، إذا تصرَّف مثل Matrix. كما يغلِّف الباني دالة element لتبديل إحداثيات القيم أسفل خط القطر، وذلك لضمان تماثل المصفوفة.

يُستخدَم super مرةً أخرى من التابع set، وذلك لاستدعاء تابع معين من مجموعة توابع الصنف الأب؛ كما سنعيد تعريف set لكن لن نستخدم السلوك الأصلي، حيث لن ينجح استدعاؤه بسبب إشارة this.set إلى set الجديد، كذلك يوفر super الواقع داخل توابع الصنف، طريقةً لاستدعاء التوابع كما عُرِّفت في الصنف الأب.

وتسمح لنا الوراثة ببناء أنواع بيانات مختلفة من أنواع موجودة مسبقًا بقليل من الجهد، وهذه -أي الوراثة- جزء أساسي في ثقافة البرمجة كائنية التوجه جنبًا إلى جنب مع التغليف وتعددية الأشكال، لكن لأن هذين الآخرَين يُعتد بهما كثيرًا في البرمجة على أساس أساليب مهمة ومفيدة، فإنّ الوراثة قد صارت محل نظر، ففي حين يُستخدَم كل من التغليف وتعددية الأشكال في فصل أجزاء الشيفرات عن بعضها مما يقلل من تعقيد البرنامج عمومًا، فالوراثة على العكس من ذلك، إذ تربط الأصناف معًا منشِئةً مزيدًا من التعقيد، لأن عليك في الغالب معرفة كيفية عمل ذلك الصنف حين تحتاج إلى الوراثة منه، بخلاف إن لم تفعل شيئًا سوى استخدامه.

وإننا نستخدمه بين الحين والآخر في برامجنا، لكن لا يحملنك ذلك على التفكير فيه أول شيء، فليس من الحكمة جعل بناء هرميات من الأصناف (شجرة عائلة من الأصناف) خيارك الأول في حل المشاكل.

6.13 عامل instanceof

توفر جافاسكربت عاملًا ثنائيًا يسمى `instanceof`، حيث نستخدمه إذا أردنا معرفة إن كان الكائن مشتقًا من صنف بعينه.

```
console.log(
  new SymmetricMatrix(2) instanceof SymmetricMatrix);
// → true
console.log(new SymmetricMatrix(2) instanceof Matrix);
// → true
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);
// → false
console.log([1] instanceof Array);
// → true
```

سينظر العامل في الأنواع المكتسبة، وسيجد أن `symmetricMatrix` نسخةٌ من `Matrix`، كما يمكن استخدام العامل مع البواني القياسية مثل `Array`، فكل كائن تقريبًا ما هو إلا نسخة من `Object`.

6.14 خاتمة

لقد رأينا أنّ نطاق تأثير الكائنات يتعدى حمل خصائصها، إذ لها نماذج أولية -والتي بدورها كائنات أيضًا- وتتصرف كما لو كان لديها خصائص ليست لديها على الحقيقة طالما أن النموذج الأولي به تلك الخصائص، كما تمكّنا من معرفة الكائنات البسيطة لها `Object.prototype` على أساس نموذج أولي لها.

يمكن استخدام البواني -وهي دوال تبدأ أسماؤها بحرف إنجليزي كبير- مع عامل `new` لإنشاء كائنات جديدة، وسيكون النموذج الأولي للكائن هو الكائن الموجود في خاصية `prototype` للباني، ونستطيع الاستفادة من

ذلك بوضع جميع الخصائص التي تشاركها القيم المعطاة -من النوع نفسه- في نماذجها الأولية. كذلك عرفنا صيغة `class` التي توفر طريقةً واضحةً لتعريف الباني ونموذجه الأولي.

تستطيع تعريف الجالبات والضابطات لاستدعاء التوابع سرًا في كل مرة يصل فيها إلى خاصية كائن ما، وقد عرفنا أن التوابع الساكنة ما هي إلا توابع مخزَّنة في باني الصنف بدلًا من نموذجه الأولي، ثم شرحنا كيف أن عامل `instanceof` يستطيع إخبارك إن أعطيته كائنًا وبانيًا، وما إذا كان الكائن نسخةً من الباني أم لا.

واعلم أنك تستطيع باستخدام الكائنات تحديد واجهة لها، وتخبر جميع أجزاء الشيفرة بأن عليهم التحدث إلى كائنك من خلال تلك الواجهة فقط، كما تُغلَّف بقية التفاصيل التي يتكون منها الكائن وتختفي خلف الواجهة. ويمكن لأكثر من نوع استخدام تلك الواجهة، فتعرف الشيفرة التي كُتبت لتستخدِم واجهةً ما كيف تعمل مع أي عدد من الكائنات المختلفة التي توفر الواجهة تلقائيًا ، وهذا ما يسمى بتعددية الأشكال.

حين تستخدم عدة أصناف لا تختلف فيما بينها إلا في بعض التفاصيل، فيمكن كتابة أصناف جديدة منها على أساس أصناف فرعية، ترث جزءًا من سلوكها.

6.15 تدريبات

6.15.1 النوع المتجهي

اكتب الصنف `Vec` الذي يمثل متجهًا في فضاء ثنائي الأبعاد، حيث يأخذ المعامِلين `x`، و`y` -وهما أرقام- ويحفظهما في خصائص بالاسم نفسه. أعطِ النموذج الأولي للصنف `Vec` تابعَين، هما: `plus`، و`minus`، اللذان يأخذان متجهًا آخر على أساس معامِل، ويُعيدان متجهًا جديدًا له مجموع قيم x، وy للمتجهين (`this`، والمعامِل)؛ أو الفرق بينهما. أضف الخاصية الجالبة `length` إلى النموذج الأولي الذي يحسب طول المتجه، وهو المسافة بين النقطة (x,y) والإحداثيات الصفرية (0,0).

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// ضع شيفرتك هنا.

console.log(new Vec(1, 2).plus(new Vec(2, 3)));
// → Vec{x: 3, y: 5}
console.log(new Vec(1, 2).minus(new Vec(2, 3)));
// → Vec{x: -1, y: -1}
console.log(new Vec(3, 4).length);
// → 5
```

إرشادات الحل

- إذا لم تكن تعرف كيف تبدو تصريحات `class`، فانظر إلى مثال صنف `Rabbit`.
- يمكن إضافة خاصية جالبة إلى الباني من خلال وضع كلمة `get` قبل اسم التابع، ولحساب المسافة من (0,0) إلى (x,y)، فيمكن استخدام نظرية فيثاغورث التي تقول: أن مربع المسافة التي نريدها يساوي مجموع مربعي x و y، وعلى ذلك يكون √(x2 + y2) هو العدد الذي نريده، ويُحسَب الجذر التربيعي في جافاسكربت باستخدام `Math.sqrt`.

6.15.2 المجموعات

توفر بيئة جافاسكربت القياسية هيكل بيانات اسمه `Set`، إذ يحمل مجموعةً من القيم مثل نسخة من `Map`، لكن على عكس `Map` فهو لا يربط قيمًا أخرى بها، بل يتتبع القيم ليعرف أيها تكون جزءًا من المجموعة. ولا يمكن للقيمة الواحدة أن تكون جزءًا من مجموعة ما أكثر من مرة واحدة، ولا يحدث أي تأثير حين تضاف مرةً أخرى.

اكتب صنفًا اسمه `Group` -بما أنّ `Set` مأخوذ من قبل- واجعل له التوابع الآتية: `add`، و`delete`، و`has`، ليكون مثل Set، حيثما ينشئ بانيه مجموعةً فارغةً، ويضيف `add` قيمةً إلى المجموعة فقط إن لم تكن عضوًا بالفعل في المجموعة، كما يحذف `delete` وسيطه من المجموعة إن كان عضوًا فيها، ويعيد `has` قيمة بوليانية توضح هل وسيطه عضو في المجموعة أم لا.

استخدم عامِل `===`، أو شيئًا يحاكيه مثل `indexof`، لمعرفة ما إذا كانت قيمتان متطابقين، وأعط الصنف التابع الساكن `from` الذي يأخذ كائنًا قابلًا للتكرار على أساس وسيط، كما ينشئ مجموعةً تحتوي على جميع القيم المنتَجة من خلال التكرار عليها.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
class Group {
  // ضع شيفرتك هنا.
}

let group = Group.from([10, 20]);
console.log(group.has(10));
// → true
console.log(group.has(30));
// → false
group.add(10);
group.delete(10);
```

```
console.log(group.has(10));
// → false
```

إرشادات الحل

تكون الطريقة الأسهل لحل هذا التدريب بتخزين مصفوفة من أعضاء المجموعة في خاصية لإحدى النُسَخ، ويمكن استخدام التابع `includes`، أو `indexOf` للتحقق من وجود قيمة ما في المصفوفة.

ويمكن لباني الصنف الخاص بك إسناد تجميعة الأعضاء إلى مصفوفة فارغة، وعند استدعاء `add` فيجب التحقق هل القيمة المعطاة موجودة في المصفوفة أم يضيفها باستخدام `push` مثلًا.

قد يكون حذف عنصر من مصفوفة في `delete` مبهمًا قليلًا، لكن تستطيع استخدام `filter` لإنشاء مصفوفة جديدة بدون القيمة، ولا تنس كتابة النسخة الجديدة من المصفوفة لتحل محل الخاصية التي تحمل الأعضاء.

يمكن للتابع `from` استخدام حلقة `for/of` التكرارية للحصول على القيم من الكائن القابل للتكرار، ويستدعي `add` لوضعها في مجموعة منشأة حديثًا.

6.15.3 المجموعات القابلة للتكرار

أنشئ الصنف `Group` من التدريب السابق، واستعن بالقسم الخاص بواجهة المكرر من هذا الفصل إن احتجت إلى رؤية الصيغة الدقيقة للواجهة.

إذا استخدمت مصفوفةً لتمثيل أعضاء المجموعة، فلا تُعِد المكرِّر المنشَأ باستدعاء التابع `Symbol.iterator` على المصفوفة، فهذا وإن كان سيعمل بدون مشاكل، إلا أنه سينافي الهدف من التدريب.

لا بأس إن تصرَّف المكرر الخاص بك تصرفًا غير مألوف عند تعديل المجموعة أثناء التكرار.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// ضع شيفرتك هنا، والشيفرة التي من المثال السابق.
for (let value of Group.from(["a", "b", "c"])) {
  console.log(value);
}
// → a
// → b
// → c
```

إرشادات الحل

ربما من الأفضل تعريف صنف `GroupIterator` جديد، كما يجب أن يكون لنُسَخ المكرر خاصية تتبع الموضع الحالي في المجموعة، بحيث تتحقق في كل مرة يُستدعى فيها `next` مما إذا كانت قد انتهت أم لا، فإن لم تنته فستتحرك متجاوزةً القيمة الحالية وتعيدها.

يحصل الصنف `Group` على تابع يسمى من قِبل `Symbol.iterator`، ويعيد عند استدعائه نسخةً جديدةً من صنف المكرر لتلك المجموعة.

6.15.4 استعارة تابع

ذكرنا أعلاه هنا أن `hasOwnProperty` لكائن يمكن استخدامه على أساس بديل قوي لعامِل `in` إذا أردت تجاهل خصائص النموذج الأولي، لكن ماذا لو كانت خارطتك map تحتاج إلى كلمة `hasOwnProperty`؟ لن تستطيع حينها استدعاء هذا التابع بما أن خاصية الكائن تخفي قيمة التابع.

هل تستطيع التفكير في طريقة لاستدعاء `hasOwnProperty` على كائن له خاصية بهذا الاسم؟ تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
let map = {one: true, two: true, hasOwnProperty: true};

// أصلح هذا الاستدعاء
console.log(map.hasOwnProperty("one"));
// → true
```

إرشادات الحل

تذكّر أن التوابع الموجودة في الكائنات المجردة تأتي من `Object.prototype`، كما تستطيع استدعاء دالة مع رابطة `this` خاصة من خلال استخدام التابع `call`.

دورة تطوير واجهات المستخدم

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

7. مشروع الروبوت

> إن السؤال عن تفكير الآلات يشبه تمامًا سؤالك هل تستطيع الغواصات السباحة أم لا.
>
> — إدزجر ديكسترا Edsger Dijkstra، المخاطر التي تهدد علوم الحاسوب.

يختلف هذا الفصل قليلًا عن باقي الفصول، إذ سأدَع الحديث عن النظريات قليلًا لننشئ نحن وأنت برنامجًا سويًا، فالنظرية وإن كانت مهمة جدًا لتعلم كيفية البرمجة، إلا أنّ قراءة البرامج الحقيقية وفهم شيفراتها لا تقل أهمية، ومشروعنا في هذا الفصل هو بناء روبوت ينفِذ مهمة في عالم افتراضي، وستكون تلك المهمة توصيل الطرود واستلامها.

7.1 قرية المرج Meadowfield

سيعمل الروبوت الخاص بنا في قرية صغيرة تُدعى قرية المرج، حيث يوجد فيها أحد عشر مكانًا بينها أربعة عشر طريقًا، ويمكن وصف القرية بالمصفوفة التالية:

```
const roads = [
  "Salma's House-Omar's House",   "Salma's House-Cabin",
  "Salma's House-Post Office",   "Omar's House-Town Hall",
  "Sara's House-Mostafa's House", "Sara's House-Town Hall",
  "Mostafa's House-Sama's House", "Sama's House-Farm",
  "Sama's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```

وتكوّن شبكة الطرق في القرية مخططًا graph، وهو عبارة عن تجميعة من نقاط لتمثل الأماكن في القرية، مع خطوط تصل بينها بحيث تمثل الطرق، وسيكون هذا هو العالم الذي سيتحرك الروبوت فيه.

لكن ليس من السهل التعامل مع مصفوفة السلاسل النصية السابقة، إذ لا نريد إلا الوجهات التي يمكننا الذهاب إليها من أي نقطة معطاة لنا، وعلى ذلك فسنحوِّل قائمة الطرق إلى هيكل بيانات يخبرنا بالمناطق التي نستطيع الذهاب إليها من كل نقطة أو مكان، انظر كما يلي:

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
```

```
  return graph;
}

const roadGraph = buildGraph(roads);
```

تُنشئ دالة `buildGraph` كائن خارطة object map عند إعطائها مصفوفة من الحدود edges، حيث تخزن فيها لكل عقدةٍ مصفوفةً من العقد المتصلة بها. كما تستخدِم تابع `method` للذهاب من سلسلة الطريق النصية التي تحمل الصيغة `"Start-End"` إلى مصفوفات من عنصرين تحتوي على البداية والنهاية على أساس سلسلتَين منفصلتين.

7.2 المهمة

سيتحرك الروبوت داخل القرية، إذ لديه طرود في أماكن مختلفة فيها، حيث يحمل كل طرد عنوانًا يجب نقله إليه؛ وسيلتقط الروبوت الطرودَ حين وصوله إليها، ثم يتركها عند وصوله إلى الوجهة المرسَل إليها، ويجب أن يقرر في كل نقطة وجهته التالية، ولا تنتهي مهمته إلا عند توصيل جميع الطرود.

لكن أولًا يجب تعريف عالم افتراضي يصف هذه العملية، وذلك لنتمكّن من محاكاته، ويكون هذا النموذج قادر على إخبارنا بموقع الروبوت والطرود معًا، كما يجب تحديث هذا النموذج عندما يقرر الروبوت الذهاب إلى مكان جديد.

إن كنت تفكر بأسلوب البرمجة كائنية التوجه، سيكون دافعك الأول هو البدء بتعريف كائنات للعناصر المختلفة في هذا العالم الذي أنشأناه، فصنف للروبوت، وآخر للطرد، وصنف ثالث للأماكن ربما، ثم تحمِل هذه الأصناف بعد ذلك خصائص تصف حالتها، كما في حالة كومة الطرود عند موقع ما، والتي يمكننا تغييرها عند إجراء تحديث لهذا العالم.

لكن هذا خطأ في أغلب الحالات على الأقل، فكون شيء ما يحاكي الكائن لا يعني وجوب معاملته على أساس كائن في برنامجك؛ كما أنّ كتابة الأصناف في برنامجك دون داعي حقيقي لها، ستُنشئ تجميعةً من الكائنات المرتبطة ببعضها بعضًا لكل منها حالة داخلية متغيرة، وتكون مثل تلك البرامج صعبة الفهم وسهلة التعطل.

لدينا أسلوبًا أفضل من ذلك، وهو ضغط حالة القرية إلى أقل فئة ممكنة من القيم التي تعرِّفها، فيكون لدينا الموقع الحالي للروبوت، وتجميعة الطرود غير المسلَّمة، والتي يحمل كل منها موقعه الحالي وعنوان التسليم، وحسبنا هذا!

بينما نحن في ذلك، فعلينا التأكد أنه حين يتحرك الروبوت من موقعه، فعلينا حساب حالة جديدة وفق الموقف الذي يكون بعد التحرك، وذلك دون تغيير الحالة الأولى، انظر كما يلي:

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place != this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place != p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

يتحقق التابع `move` أولًا إن كان ثمة طريق من الموقع الحالي إلى موقع الوِجهة، وإن لم يكن، فسيعيد الحالة القديمة بما أنّ هذه الخطوة غير صالحة، ثم يُنشئ حالةً جديدةً يكون فيها موقع الوِجهة هو الموقع الجديد للروبوت.

لكن سيحمل هذا الروبوت معه طرودًا أخرى غير هذا الطرد، ويأخذها معه إلى موقع تسليم الطرد المسمى، ثم يتابع حملها معه بعد تسليم الطرد، ليذهب بكل منها إلى موقع تسليمه الذي يخصه، ولكي نحصل على بيانات تلك الطرود عند أي نقطة زمنية نريدها، فسنحتاج إلى إنشاء فئة set جديدة نضع فيها تلك الطرود التي يحملها إلى الموقع الجديد، ثم يترك الطرود الواجب تسليمها في موقع التسليم، أي نحذفها من فئة الطرود غير المسلَّمة. ويتكفل بعملية الانتقال استدعاء `map`، بينما نستدعي `filter` ليتولى عملية التسليم.

لا تتغير كائنات الطرود عند نقلها، وإنما يعاد إنشاؤها، ويعطينا التابع `move` حالة جديدة للقرية مع ترك الحالة القديمة كما هي دون تغيير، انظر كما يلي:

```
let first = new VillageState(
  "Post Office",
  [{place: "Post Office", address: "Salma's House"}]
);
let next = first.move("Salma's House");
```

```
console.log(next.place);
// → Salma's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
```

تُسلَّم الطرود عند مواضع تسليمها مع حركة الروبوت بين تلك المواقع، ويُرى أثر ذلك في الحالة التالية، لكن ستظل الحالة الابتدائية تصف الموقف الذي يكون فيه الروبوت عند مكتب البريد ومعه الطرد الغير مسلَّم بعد.

7.3 البيانات الثابتة Persistent Data

تُسمى هياكل البيانات التي لا تتغير بالهياكل الثابتة persistent، أو غير القابلة للتغير immutable، وتحاكي السلاسل النصية والأرقام في بقائها كما هي، بدلًا من احتواء أشياء مختلفة في كل مرة.

بما أن كل شيء في جافاسكربت قابل للتغير تقريبًا، فسيتطلب العمل مع كائنات أو بيانات ثابتة جهدًا وعملًا إضافيًا ونوعًا من التقييد، حيث لدينا دالة `Object.freeze` من أجل هذا، إذ تؤثر على الكائن وتجمده، وذلك لتجعله يتجاهل الكتابة على خصائصه، فتستطيع استخدام ذلك لضمان ثبات كائناتك إن أردت، لكن تذكر أن هذا التجميد يعني أن الحاسوب سيبذل مزيدًا من الجهد.

نفضل إخبار العامة بترك الكائن الفلاني وشأنه وعدم العبث به، آملين تذكّرهم ذلك، بدلًا من إخبارهم بتجاهل تحديثات بعينها.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

لكن هنا يبرز سؤال إن كنت منتبهًا، فإن كانت اللغة نفسها تحثنا على جعل كل شيء متغيرًا وتتوقع منا ذلك، فلماذا نخرج عن هذا المسلك لنجعل بعض الكائنات ثابتةً لا تقبل التغيير؟

الإجابة بسيطة وهي أن هذا سيساعدنا في فهم برامجنا أكثر، إذ يتعلق الأمر بإدارة التعقيد لبرامجنا، فإن كانت الكائنات التي في نظامنا ثابتةً ومستقرةً، فيمكننا إجراء عمليات عليها إجراءً معزولًا -حيث سيعطينا التحرك إلى منزل سلمى مثلًا من أي حالة بدء معطاة الحالة الجديدة نفسها في كل مرة-؛ أما إن كانت الكائنات تتغير مع الوقت، فسيضيف هذا بعدًا جديدًا من التعقيد إلى العمليات والتفكير المنطقي لحل المشكلة.

وربما تقول أن مسألة توصيل الطرود، والروبوت، وهذه القرية الصغيرة، سهل أمرها ويمكن إدارتها، وربما أنت محق، لكن المعيار الذي يحدد نوع النظم الممكن بناؤها هو فهمنا نحن لتلك الأنظمة ومدى ما يمكننا فهمه مطلقًا، فأيّ شيء يسرّع فهم شيفرتك، فسيمهد لك الطريق لبناء نظم أكثر تطورًا.

لكن رغم سهولة فهم النظم المبنية على هياكل بيانات ثابتة، فقد يكون من الصعب تصميم نظام مثل هذا، وخاصةً إن لم تكن لغة البرمجة التي تستخدمها مصمَّمةً لذلك، كما سنبحث عن فرص استخدام هياكل البيانات الثابتة في هذا الكتاب مع استخدامنا للهياكل المتغيرة كذلك.

7.4 المحاكاة

ينظر روبوت التوصيل إلى العالم، ويقرر الاتجاه الذي يريد السير فيه، وعليه فيمكننا القول أنّ الروبوت هو دالة تأخذ كائن `villageState`، وتعيد اسم مكان قريب.

لأننا نريد الربوتات أن تكون قادرة على تذكر أشياء بعينها كي تنفِّذ الخطط الموضوعة لها من قِبلنا، فسنمرر إليها ذاكرتها، ونسمح لها بإعادة ذاكرة جديدة، ومن ثم يعيد الروبوت كائنًا يحتوي الاتجاه الذي يريد السير فيه، وقيمة ذاكرة تُعطى إليه في المرة التالية التي يُستدعى فيها.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Done in ${turn} turns`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Moved to ${action.direction}`);
  }
}
```

لننظر ما الذي يجب على الروبوت فعله كي "يحل" حالة ما:

يجب عليه التقاط جميع الطرود أولًا من خلال الذهاب إلى كل موقع فيه طرد، ثم يسلِّم تلك الطرود بالذهاب إلى كل عنوان من العناوين المرسل إليها هذه الطرود، حيث لا يذهب إلى موقع التسليم إلا بعد التقاط الطرد الخاص به.

تُرى ما هي أغبى خطة يمكن أن تنجح هنا؟ إنها العشوائية لا شك، حيث يتخِذ الروبوت اتجاهًا عشوائيًا عند كل منعطف، مما يعني أنه سيمر لا محالة على جميع الطرود بعد عدد من المحاولات، وسيصل كذلك إلى موقع تسليمها في مرحلة ما. انظر:

```
function randomPick(array) {
  let choice = Math.floor(Math.random() * array.length);
  return array[choice];
}

function randomRobot(state) {
  return {direction: randomPick(roadGraph[state.place])};
}
```

تعيد `Math.random()` عددًا بين الصفر والواحد، ويكون دومًا أقل من الواحد، كما يعطينا ضرب عدد مثل هذا في طول أي مصفوفة ثم تطبيق `Math.floor` عليه موقعًا عشوائيًا للمصفوفة. وبما أنّ هذا الروبوت لا يحتاج إلى تذكر أي شيء، فسيَتجاهل الوسيط الثاني له ويهمل خاصية `memory` في كائنه المعاد؛ وتذكّر أنّه يمكن استدعاء دوال جافاسكربت بوسائط إضافية دون آثار جانبية مقلقة.

نحتاج الآن إلى طريقة لإنشاء حالة جديدة بها بعض الطرود، وذلك من أجل إطلاق هذا الروبوت للعمل، والتابع الساكن -المكتوب هنا بإضافة خاصيةٍ مباشرةً للمنشئ- مكان مناسب لوضع هذه الوظيفة.

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
  }
  return new VillageState("Post Office", parcels);
};
```

لا نريد أي طرود مرسلة من المكان نفسه الذي ترسَل إليه، ولهذا تختار حلقة `do` التكرارية أماكنًا جديدةً في كل مرة تحصل على مكان مطابق للعنوان.

دعنا نبدأ عالمًا افتراضيًا، كما يلي:

```
runRobot(VillageState.random(), randomRobot);
// → Moved to Marketplace
// → Moved to Town Hall
// → …
// → Done in 63 turns
```

كما نرى من المثال، إذ غيّر الروبوت اتجاهه ثلاثًا وستين مرة، وذلك الرقم الكبير بسبب عدم التخطيط الجيد للخطوة التالية، كما سننظر في هذا قريبًا.

يمكنك استخدام دالة `runRobotAnimation` المتاحة في البيئة البرمجية لهذا الفصل، حيث ستنفِّذ المحاكاة، وستعرض لك الروبوت وهو يتحرك في خارطة القرية، بدلًا من إخراج نص فقط.

```
runRobotAnimation(VillageState.random(), randomRobot);
```

سندع طريقة تنفيذ `runRobotAnimation` مبهمةً في الوقت الحالي، حيث ستستطيع معرفة كيفية عملها بعد قراءة الفصل الرابع عشر من هذا الكتاب والذي نناقش فيه تكامل جافاسكربت في المتصفحات.

7.5 طريق شاحنة البريد

لا شك أنّ فكرة الروبوت هذه لتوصيل البريد فكرة بدائية، وأجدر بنا تطوير هذا البرنامج قليلًا، فلم لا ننظر إلى توصيل البريد في العالم الحقيقي لنستوحي منه أفكارًا لعالمنا الصغير؟

أحد هذه الحلول هو البحث عن طريق يمر على جميع الأماكن في القرية، وحينها يأخذ الروبوت هذه الطريق مرتين. انظر مثلًا على هذا الطريق بدءًا من مكتب البريد:

```
const mailRoute = [
  "Salma's House", "Cabin", "Salma's House", "Omar's House",
  "Town Hall", "Sara's House", "Mostafa's House",
  "Sama's House", "Shop", "Sama's House", "Farm",
  "Marketplace", "Post Office"
];
```

نحتاج إلى الاستفادة من ذاكرة الروبوت إذا أردنا استخدام الروبوت المتتبع للطريق route-following، حيث يحذف الروبوت العنصر الأول عند كل منعطف ويحتفظ ببقية الطريق:

```
function routeRobot(state, memory) {
  if (memory.length == 0) {
    memory = mailRoute;
  }
  return {direction: memory[0], memory: memory.slice(1)};
}
```

سيكون هذا الروبوت بهذا الأسلوب الجديد أسرع من الأسلوب الأول، إذ سيكون أقصى عدد من الاتجاهات التي يسلكها أو المنعطفات التي يأخذها هو 26 -وهو ضعف عدد الأماكن في طريق القرية-، وإن كان في العادة أقل من هذا العدد، فليست جميع الأماكن بها طرود بريد في كل مرة.

7.6 اكتشاف الطريق Pathfinding

لنبحث في طريقة أكثر تطورًا من المرور على جميع الأماكن في القرية في كل مرة، فلا شك في أن الروبوت سيكون أكثر كفاءة إذا عدّل سلوكه ليناسب العمل الحقيقي المراد تنفيذه؛ لذا يجب امتلاكه القدرة على التحرك نحو طرد معين أو موقع تسليم بعينه، وعليه نحتاج دالة لإيجاد الطريق المناسبة.

تُعَدّ مشكلة إيجاد الطريق باستخدام مخطط هي مشكلة البحث، حيث نستطيع تحديد ما إذا كان الحل المعطى -أي الطريق- مناسبًا أم لا، لكن لا نستطيع حساب الحل بالطريقة نفسها عند حساب حاصل جمع 2+2 مثلًا، وإنما يجب إنشاء حلولًا محتملةً لإيجاد واحد صالح.

تتضح هنا المشكلة أكثر، إذ لا نهايةً لعدد الاحتمالات الممكنة للطرق من خلال مخطط، لكن يمكن تضييق عدد الاحتمالات إذا أردنا الطرق المؤدية من نقطة أ إلى نقطة ب مثلًا، فعندئذ لن يهمنا سوى الطرق التي تبدأ من النقطة أ، كما لا نريد الطرق التي تمر المكان نفسه مرتين، إذ لا تُعَدّ هي الطرق الأكثر كفاءة في أي مكان، ويقلل هذا عدد الطرق التي يجب اعتمادها من قِبَل دالة البحث أكثر وأكثر.

نريد في الواقع أقصر طريق فقط، وعليه يجب النظر في الطرق الأقصر أولًا قبل النظر في الطرق الأطول، وأحد الأساليب لفعل ذلك هو بدء طرق من نقطة تحرك الروبوت لتُستكشف جميع الأماكن التي يمكن الوصول إليها ولم يُذهب إليها بعد، حتى تصل إحدى تلك الطرق إلى المكان المراد تسليم الطرد إليه، وهكذا نستكشف الطرق التي يحتمل أن تكون مفيدة لنا، ونتخذ أقصرها أو واحدًا من أقصر تلك الطرق. كما في الدالة التالية:

```
function findRoute(graph, from, to) {
  let work = [{at: from, route: []}];
  for (let i = 0; i < work.length; i++) {
    let {at, route} = work[i];
    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
```

```
        if (!work.some(w => w.at == place)) {
          work.push({at: place, route: route.concat(place)});
        }
      }
    }
  }
```

يجب أن يتم الاستكشاف بترتيب صحيح، فالأماكن التي يصل إليها أولًا يجب استكشافها أولًا، لكن انتبه إلى أن مكان س مثلًا لا يُستكشف فور الوصول إليه، إذ سيعني هذا أنه علينا استكشاف أماكن يمكن الوصول إليها من س وهكذا، رغم احتمال وجود مسارات أقصر لم تُستكشف بعد.

وعلى هذا فإن الدالة تحتفظ بقائمة عمل work list، وهي مصفوفة من الأماكن التي يجب استكشافها فيما بعد مع الطرق التي توصلنا إليها، حيث تبدأ بموقع ابتدائي للبدء منه وطريق فارغة، ثم يبدأ البحث بعدها بأخذ العنصر التالي في القائمة واستكشافه، مما يعني النظر في جميع الطرق الخارجة من هذا المكان، فإن كان أحدها هو الهدف المراد، فيُعاد طريق تامة finished road، وإلا فسيضاف عنصر جديد إلى القائمة إن لم يُنظر في هذا المكان من قبل.

كذلك، إن كنا قد نظرنا فيه من قبل، فقد وجدنا إما طريق أطول إلى هذا المكان أو واحدة بنفس طول الطريق الموجودة -وبما أننا نبحث عن الطرق الأقصر أولًا- ولا نحتاج إلى استكشافها عندئذ.

تستطيع تخيل هذا بصريًا على أساس شبكة من الطرق المعروفة، حيث تخرج من موقع ابتدائي، وتنمو بانتظام في جميع الجوانب، لكن لا تتشابك مع بعضها أو تنعكس على نفسها؛ وبمجرد وصول خيط من تلك الشبكة إلى الموقع الهدف، فيُتتبَّع ذلك الخيط إلى نقطة البداية ليعطينا الطريق التي نريدها.

لا تعالج شيفرتنا الموقف الذي لا تكون فيه عناصر عمل على قائمة العمل، وذلك لِعلمنا باتصال المخطط الخاص بنا، أي يمكن الوصول إلى كل موقع من جميع المواقع الأخرى، وسنكون قادرين دائمًا على إيجاد طريق ممتدة بين نقطتين، فلا يفشل البحث الذي نجريه.

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
```

```
  return {direction: route[0], memory: route.slice(1)};
}
```

يستخدِم هذا الروبوت قيمة ذاكرته على أساس قائمة من الاتجاهات ليتحرك وفقًا لها، تمامًا مثل الروبوت المتتبع للطريق الذي ذكرناه آنفًا، وعليه معرفة الخطوة التالية إذا وجد القائمة فارغة؛ كما ينظر في أول طرد غير مسلَّم في الفئة التي معه، فإن لم يكن قد التقطه، فسيرسم طريقًا إليه، وإن كان قد التقطه، فسينشئ طريقًا إلى موقع التسليم، انظر كما يلي:

```
runRobotAnimation(VillageState.random(),
                  goalOrientedRobot, []);
```

ينهي هذا الروبوت مهمة تسليم خمسة طرود في نحو ستة عشر منعطفًا، وهذا أفضل بقليل من `routeRobot`، لكن لا زال هناك فرصة للتحسين أكثر من ذلك.

7.7 تدريبات

7.7.1 معايرة الروبوت

من الصعب موازنة الروبوتات بجعلها تحل بعض السيناريوهات البسيطة، فلعل أحدها حصل على مهام سهلة دون الآخر.

اكتب دالة `compareRobots` التي تأخذ روبوتين مع ذاكرتهما الابتدائية، وتولد مئة مهمة، ثم اجعل كل واحد من الروبوتين يحل هذه المهام كلها واحدة واحدة، ويجب أن يكون الخرج عند انتهائهما هو العدد المتوسط للخطوات التي قطعها كل واحد لكل مهمة.

تأكد من إعطاء المهمة نفسها لكلا الروبوتين في تلك المهام المئة لضمان العدل والدقة في النتيجة، بدلًا من توليد مهام مختلفة لكل روبوت.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
function compareRobots(robot1, memory1, robot2, memory2) {
  // شيفرتك هنا
}

compareRobots(routeRobot, [], goalOrientedRobot, []);
```

إرشادات الحل

سيكون عليك كتابة صورة من دالة `runRobot`، بحيث تعيد عدد الخطوات التي قطعها الروبوت لإتمام المهمة، بدلًا من تسجيل الأحداث في الطرفية.

عندئذ تستطيع دالة القياس توليد حالات جديدة في حلقة تكرارية، وعدّ خطوات كل روبوت؛ وحين تولد قياسات كافية، فيمكنها استخدام `console.log` لإخراج المتوسط لكل روبوت، والذي سيكون ناتج قسمة العدد الكلي للخطوات المقطوعة على عدد القياسات.

7.7.2 كفاءة الروبوت

هل تستطيع كتابة روبوت ينهي مهمة التوصيل أسرع من `goalOrientedRobot`؟ ما الأشياء التي تبدو غبيةً بوضوح؟ وكيف يمكن تطويرها؟

إن حللت التدريبات السابقة، فربما تود استخدام دالة `compareRobots` التي أنشأتَها قبل قليل للتحقق إن كنت قد حسّنت الروبوت أم لا.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// شيفرتك هنا

runRobotAnimation(VillageState.random(), yourRobot, memory);
```

إرشادات الحل

إن القيد الرئيسي لـ `goalOrientedRobot` هو تعاملها مع طرد واحد في كل مرة، وستمضي ذهابًا وإيابًا في القرية، وذلك لأن الطرد الذي تريده موجود على الناحية الأخرى من الخارطة، حتى لو كان في طريقها طرود أخرى أقرب.

أحد الحلول الممكنة هنا هو حساب طرق جميع الطرود ثم أخذ أقصرها، ويمكن الحصول على نتائج أفضل إذا كان لدينا عدة طرق قصيرة، فسنختار حينها الطرق التي فيها التقاط طرد بدلًا من التي فيها تسليم طرد.

7.7.3 المجموعة الثابتة

ستجد أغلب هياكل البيانات الموجودة في بيئة جافاسكربت القياسية لا تناسب الاستخدام الثابت، حيث تملك المصفوفات التابعَين `slice` و`concat` اللذَين يسمحان لنا بإنشاء مصفوفات جديدة دون تدمير القديمة، لكن `Set` مثلًا ليس فيه توابع لإنشاء فئة جديدة فيها عنصر مضاف إلى الفئة الأولى أو محذوف منها.

اكتب صنف جديد باسم PGroup يشبه الصنف Group من الفصل السادس، حيث يخزن مجموعة من القيم، وتكون له التوابع add، وdelete، وhas، كما في الصنف Group تمامًا.

يعيد التابع add فيه نسخةً جديدةً من PGroup مع إضافة العضو المعطى given member وترك القديم دون المساس به، وبالمثل، فيجب على التابع delete إنشاء نسخةً جديدةً ليس فيها العضو المعطى.

يجب أن يعمل هذا الصنف مع أي نوع من القيم وليس السلاسل النصية فقط، ولا تُشترط كفاءته عند استخدامه مع كميات كبيرة من القيم؛ كذلك ليس شرطًا أن يكون الباني constructor جزءًا من واجهة الصنف رغم أنك تريد استخدام ذلك داخليًا، وإنما هناك نسخة فارغة PGroup.empty يمكن استخدامها على أساس قيمة ابتدائية.

لماذا تظن أننا نحتاج إلى قيمة PGroup.empty واحدة فقط بدلًا من دالة تنشئ خارطةً جديدةً وفارغةً في كل مرة؟

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
class PGroup {
  // شيفرتك هنا
}

let a = PGroup.empty.add("a");
let ab = a.add("b");
let b = ab.delete("a");

console.log(b.has("b"));
// → true
console.log(a.has("b"));
// → false
console.log(b.has("a"));
// → false
```

إرشادات الحل

ستكون أفضل طريقة لتمثيل مجموعة من القيم الأعضاء مصفوفةً لسهولة نسخها. تستطيع إنشاء مجموعةً جديدةً حين تضاف قيمة ما إليها، وذلك بنسخ المصفوفة الأصلية التي فيها القيمة المضافة -باستخدام concat مثلًا- وتستطيع ترشيحها من المصفوفة حين تُحذف تلك القيمة.

يستطيع باني الصنف أخذ مثل هذه المصفوفة على أساس وسيط، ويخزنها على أنها الخاصية الوحيدة للنسخة، ولا تُحدَّث هذه المصفوفة بعدها.

يجب إضافة الخاصية `empty` إلى باني غير تابع بعد تعريف الصنف، مثل وسيط عادي.

تحتاج إلى نسخة `empty` واحدة فقط بسبب تطابق المجموعات الفارغة وعدم تغير نُسَخ الصنف، كما تستطيع إنشاء مجموعات مختلفة عديدة من هذه المجموعة الفارغة دون التأثير عليها.

خمسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من $5 فقط

تصفح الخدمات

8. الزلات البرمجية والأخطاء

بات إصلاح الأخطاء وتنقيحها في الشيفرة البرمجية أصعب بمرتين من كتابة الشيفرة نفسها، ولا أرى المرء أهلًا لتنقيح الشيفرة إن كان يتذاكى في كتابتها.

— برايان كِرنيجان Brian Kernighan، وفيلِب جيمس بلوجر Phillip James Plauger، كتاب عناصر أسلوب البرمجة.

تُسمى الأخطاء في برامج الحاسوب عادةً بالزلات bugs، ونحن من نضع هذه الزلات في برامجنا بأيدينا حين نخطئ في شيء ما، أو ننسى رمزًا، أو محرفًا، أو نضع واحدًا في غير محله، وإن كان يحلو للكثير منا أن يظن بأنها تزحف من تلقاء نفسها إلى داخل الشيفرة، ولو قلنا أن البرنامج هو فكرة متبلورة في ذهن المبرمج، فاحتمالية حدوث ثغرة أو خطأ برمجي في هذا البرنامج لن تخرج من أحد شيئين:

- الفكرة نفسها معيبة أو مشوهة.
- حدوث خطأ أثناء ترجمة البرنامج من فكرة إلى شيفرة برمجية.

والحالة الثانية أيسر في اكتشافها وحلها من الأولى، إذ يكون البرنامج سليمًا عدا هذه الثغرة أو الخطأ؛ أما إن كان البرنامج كله مشوهًا نتيجة عيب في كل من الفكرة والمبدأ اللذَين بُني عليهما، فسيكون ذلك أصعب بكثير، ويُطلق على عملية اكتشاف الأخطاء وتصحيحها في الاصطلاح الأجنبي debugging.

8.1 اللغة

سبب أغلب الأخطاء التي تحدث في البرامج التي نكتبها كما ذكرنا هو نحن المبرمجين، ولو كان الحاسوب يَعقل لقذف تلك المشاكل في وجوهنا، وإذا علمت مرونة جافاسكربت العالية لأدركت مدى صعوبة تنقيح الأخطاء الموجودة في شيفراتك التي تكتبها.

تُعَدّ بنية الرابطات bindings، والخصائص properties مبهمةً إلى الحد الذي يندر معه اكتشاف الأخطاء الكتابية قبل تشغيل البرنامج، وحتى عند التشغيل، إذ تسمح لك بالقيام بأمور غير منطقية دون تنبيهك إليها مثل حساب `true * "monkey"`.

لكن رغم تلك المرونة الكبيرة في جافاسكربت، إلا أن لها حدودًا لا تتسامح معها، فمثلًا، ستجعل الحاسوب ينبهك فورًا إذا كتبت برنامجًا لا يتبع قواعدها وبنيتها اللغوية؛ كما سيُحدث استدعاء شيء ما غير الدوال أو البحث عن خاصية في قيمة غير معرفة خطأً يُرسَل في تقرير حين يحاول البرنامج تنفيذ هذا الإجراء أي عند الاستدعاء أو البحث في هاتين الحالتين.

لكن الغالب أنه لن تُنتج حساباتك الغير منطقية سوى `NaN` -أي ليس عددًا Not A Number- أو قيمة غير معرفة undefined value، وسيتابع البرنامج تنفيذه ظانًا أنه يقوم بشيء مفيد، إذ لن تظهر المشكلة إلا لاحقًا بعد مرور تلك القيمة الزائفة على عدة دوال، وقد لا تُطلق إنذار الخطأ على الإطلاق، لكنها تتسبب في خطأ الخرج الناتج من البرنامج في نفس الوقت! وبناءً على ذلك فمن الصعب العثور على مصدر مثل تلك المشاكل.

8.2 الوضع الصارم strict mode

يمكن تقييد جافاسكربت للحد من مرونتها العالية، وذلك من خلال تفعيل الوضع الصارم strict mode فيها، ويكون هذا بوضع السلسلة النصية `"use strict"` في رأس الملف أو متن الدالة، انظر مثالًا لذلك كما يلي:

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++) {
    console.log("Happy happy");
  }
}

canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

إذا نسيت وضع `let` قبل الرابطة، كما في حالة `counter` التي في المثال أعلاه، فستُنشِئ جافاسكربت رابطةً عامةً global binding وستستخدِمها؛ أما في الوضع الصارم فلا يحدث ذلك، بل تبلغك اللغة بالخطأ، وذلك أكثر فائدةً لك في البرمجة؛ لكن يجب الانتباه إلى أن هذا لا يحدث حين تكون الرابطة موجودة أصلًا على أساس رابطة عامة، ففي تلك الحالة ستظل الحلقة التكرارية تستبدل قيمة الرابطة.

كما تحمل رابطة `this` في الوضع الصارم قيمةً غير معرفة `undefined` في الدوال التي لا تُستدعى على أساس توابع methods؛ أما في الاستدعاء العادي، فستشير `this` إلى كائن النطاق العام global scope

object الذي تكون خصائصه هي الرابطات العامة، فإن استدعيت تابعًا أو بانيًا بالخطأ في الوضع الصارم، فستعطيك جافاسكربت الخطأ بمجرد محاولة قراءة شيء من this بدلًا من الكتابة في النطاق العام، فمثلًا، انظر الشيفرة التالية التي تستدعي دالة باني دون كلمة new المفتاحية كي لا تشير this فيها إلى كائن باني جديد:

```
function Person(name) { this.name = name; }
let osama = Person("Osama"); // oops
console.log(name);
// → Osama
```

ينجح هنا هذا الاستدعاء الزائف إلى person، لكنه يعيد قيمةً غير معرَّفة، وينشئ رابطة name العامة؛ أما في الوضع الصارم فستكون النتيجة مختلفةً، انظر كما يلي:

```
"use strict";
function Person(name) { this.name = name; }
let osama = Person("Osama"); // forgot new
// → TypeError: Cannot set property 'name' of undefined
```

كما ترى فقد أخبرتنا اللغة مباشرةً بوجود خطأ ما، وهذا أكثر فائدةً لنا لا ريب.

لحسن حظنا فستشتكي البواني constructors التي أُنشئت باستخدام صيغة class إذا استُدعيت من غير new، مما يجعل هذه المشكلة أقل إزعاجًا حتى في الوضع العادي أو خارج الوضع الصارم، وإضافةً إلى ما سبق، فيملك الوضع الصارم بعض الخصائص الأخرى، إذ يرفض إعطاء دالة ما عوامل متعددة بالاسم نفسه، كما يزيل بعض مزايا اللغة المسببة لمشاكل مثل تعليمة with التي لن نذكرها مرةً أخرى في هذا الكتاب لكثرة مشاكلها.

لن يؤذيك ولن يضرك استخدام الوضع الصارم عن طريق كتابة "use strict" في المجمل، وإنما ينفعك ويفيدك في اكتشاف المشاكل.

8.3 الأنواع Types

تريد بعض اللغات معرفة أنواع الرابطات والتعبيرات قبل تشغيل البرنامج، حيث ستخبرك مباشرةً إذا استُخدِم أحد الأنواع بصورة متناقضة؛ أما جافاسكربت فلا تنظر إلى الأنواع إلا عند تشغيل البرنامج، كما تحاول ضمنيًا تحويل القيم إلى الأنواع التي تتوقعها هي عادةً.

لكن مع هذا، تقدم الأنواع إطار عمل framework مفيدًا عند الحديث عن البرامج، فتَنتج أكثر الأخطاء من الجهل بنوع القيمة الداخلة إلى دالة أو الخارجة منها، فإذا كانت عندك هذه المعلومات مكتوبةً، فسيقل احتمال

حدوث تلك الأخطاء لا ريب، حيث تستطيع إضافة تعليق مثل الذي في الشيفرة التالية قبل دالة `goalOrientedRobot` التي أنشأناها في الفصل السابق لتصف نوعها:

```
// (VillageState, Array) → {direction: string, memory: Array}
function goalOrientedRobot(state, memory) {
  // ...
}
```

هناك العديد من الاصطلاحات المتّبَعة لإدخال الأنواع في برامج جافاسكربت، وتحتاج الأنواع إلى تحديد مدى تعقيدها لتستطيع وصف ما يكفي من الشيفرة بحيث يكون مفيدًا، فماذا يكون النوع الخاص بدالة `randomPick` مثلًا التي تعيد عنصرًا عشوائيًا من مصفوفة ما؟

سنحتاج إلى تحديد متغير نوع type variable، وليكن T مثلًا الذي سيمثل أي نوع، وبذلك نستطيع إعطاء `randomPick` نوعًا مثل `([T]) → T`، وهي دالة من مصفوفة مكونة من Ts إلى T.

إذا كانت الأنواع الموجودة في البرنامج معروفةً، فسيتمكن الحاسوب من التحقق منها بدلًا عنك، وذلك ليُخرج لك الأخطاء قبل تشغيل البرنامج.

هناك العديد من أشكال جافاسكربت التي تضيف الأنواع إلى اللغة وتتحقق منهم، لعل أشهرها لغة TypeScript، وننصحك بتجربتها إن كنت تريد إضافة بعض الصرامة إلى برامجك؛ أما في هذا الكتاب فسنعمل بجافاسكربت العادية.

8.4 الاختبار

إذا لم تعيننا اللغة على إيجاد الأخطاء، فيجب البحث عنهم بالطريقة الصعبة من خلال تشغيل البرنامج، وعندها سنرى إن كان سيعمل كما خططنا له أم لا، لكن تنفيذ هذا يدويًا مرةً بعد مرة ليس الطريقة المثلى للبرمجة، حيث تُعَدّ مزعجةً وغير عملية بما أنها تستغرق وقتًا طويلًا وجهدًا مضنيًا لاختبار كل شيء في كل مرة تغير فيها شيئًا واحدًا.

كما سنستغل قدرة الحواسيب الهائلة في العمليات التكرارية، بما أن الاختبار في حد ذاته عملية تكرارية، فلمَ لا نجعل هذه العملية مؤتمتةً؟ وسيكون ذلك بكتابة برنامج اختبار يتحقق من البرنامج الخاص بنا.

وقد تقف هنا لحظة لتقول ألم نرد وسيلةً لنقلل بها الجهد الواقع علينا؟ ونجيبك أن بلى، لكن الجهد الذي ستبذله مرةً واحدةً فقط في كتابة هذا الاختبار، سيغنيك طيلة العمل على البرنامج محل المشروع نفسه الذي بين يديك، وستشعر أن بيديك قوةً خارقةً لا تأخذ سوى بضع ثواني، حيث سيتحقق من عمل برنامجك بكفاءة في كل المواقف التي كتبت اختبارات لها، وستلاحظ مباشرةً فور تعطيلك لشيء ما بالخطأ، بدلًا من مرور الخطأ وعدم الشعور به إلا حين تقابله صدفةً مرةً أخرى أثناء العمل على شيء جديد.

تأخذ الاختبارات عادةً صورة برامج صغيرة معنونة لتتحقق من أجزاء بعينها في شيفرتك، فمثلًا، ستكون بعض الاختبارات القياسية لتابع `toUpperCase` -والتي لعل أحدًا غيرنا اختبرها من قبل- على الصورة التالية:

```
function test(label, body) {
  if (!body()) console.log(`Failed: ${label}`);
}

test("convert Latin text to uppercase", () => {
  return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΪΡΕΤΕ";
});
test("don't convert case-less characters", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

تُنتِج كتابة الاختبارات التي تحاكي المثال أعلاه شيفرات متكررةً وغريبةً، ولحل هذه المشكلة فلدينا برامج ستساعدك على بناء وتشغيل تجميعات مختلفة من الاختبارات (حِزم اختبارات)، وذلك من خلال توفير لغة مناسبة في شكل دوال وتوابع تناسب كتابة الاختبارات، وكذلك بإخراج معلومات مفيدة حينما يفشل أحد تلك الاختبارات، ويطلق على ذلك عادةً اسم منفِّذات الاختبارات test runners.

كلما زاد عدد الكائنات الخارجية التي تتعامل الشيفرة معها، صعُب إعداد سياق لاختبارها فيه، وقد كان أسلوب البرمجة الذي عرضناه في الفصل السابع أسهل في الاختبار، حيث استخدَم قيمًا ثابتةً persistent values عوضًا عن كائنات متغيرة.

8.5 التنقيح Debugging

إذا عرفت أن ثمة شيء في برنامجك يجعله يتصرف على نحو لم تتوقعه أو ينتج أخطاءً، فالخطوة التالية منطقيًا هي معرفة ما هو ذلك الشي أو هذه المشكلة، وقد تكون تلك المشكلة واضحةً أحيانًا، إذ تشير رسالة الخطأ إلى سطر بعينه في برنامجك، حيث سترى المشكلة إذا نظرت إلى وصف الخطأ وذلك السطر بنفسك.

لكن أحيانًا يكون السطر المسبب للمشكلة ضحيةً لاستخدام قيمة متذبذبة وغير مستقرة عليه، بحيث تكون تلك القيمة منتَجةً في مكان آخر، وتُستخدم في هذا السطر بصورة خاطئة، ومن المحتمل رؤيتك لهذا إن جربت حل التمارين التي في الفصول السابقة من هذا الكتاب.

يحول البرنامج في المثال التالي العدد الصحيح إلى سلسلة نصية في نظام ما سواءً كان ثنائيًا، أو عشريًا، أو غيرهما، وذلك بأخذ آخر رقم، ثم تقسيم العدد للتخلص من ذلك الرقم، وهكذا دواليك، لكن يشير الخرج الذي ينتجه البرنامج الآن إلى وجود خطأ ما.

```
function numberToString(n, base = 10) {
  let result = "", sign = "";
  if (n < 0) {
    sign = "-";
    n = -n;
  }
  do {
    result = String(n % base) + result;
    n /= base;
  } while (n > 0);
  return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e-3181.3…
```

لعلك انتبهت إلى المشكلة إذا نظرت إلى الشيفرة أعلاه، لكن نريدك التخيل للحظة أنك لا تعرفها ولم تلاحظها.

لنبحث في سياق الحل والتنقيح الذي يجب عمله، إذ نعلم بعدم تصرف برنامجنا على النحو الذي نريده ونريد معرفة السبب، فهنا يجب مقاومة الرغبة في إجراء تغييرات عشوائية في الشيفرة من دون تفكير مسبق، وتحليل للقرار والتغييرات التي تجريها.

نريدك الآن الوقوف للحظة، والتفكير، وتحليل الموقف وما يحدث مع البرنامج، وجمع الملاحظات حول ذلك، للخروج بنظرية حول سبب الخطأ، ثم اختبار تلك النظرية، وستكون إحدى طرق ذلك بوضع بعض استدعاءات `console.log` في البرنامج لتحصل على معلومات إضافية عما يفعله، كما نريد هنا في حالتنا لـ `n` أخذ القيم 13، و1، ثم 0.

لنكتب قيمتها في بداية الحلقة التكرارية:

```
13
1.3
0.13
0.013
```

```
…
1.5e-323
```

لا تعطي قسمة 13 على 10 عددًا صحيحًا، لذلك نريد `n = Math.floor(n / base)` بدلًا من `n /=` `base` كي يتحرك العدد إلى اليمين كما نريد.

المزايا التي توفرها أداة المنقِّح debugger والتي تأتي مدمجةً في المتصفحات هي إحدى الوسائل التي نستطيع استخدامها كي ننظر في سلوك البرنامج ونختبره، إذ تكون في هذه المتصفحات مزية إنشاء نقطة توقف breakpoint عند سطر بعينه داخل البرنامج، حيث سيتوقف تنفيذ البرنامج عند وصوله إلى ذلك السطر كي تنظر أنت في قيم الرابطات عند هذه النقطة وتفحصها، ولأن المنقِّحات تختلف من متصفح لآخر، فلن نخوض بك في تفاصيل هذه المزايا أكثر من ذلك، إذ سنترك لك حرية النظر في أدوات المطور في متصفحك، أو البحث في شبكة الويب عن مزيد من المعلومات عن هذا الأمر.

بالعودة إلى فحص سلوك البرنامج، فمن الممكن إنشاء نقطة توقف بوضع تعليمة `debugger` في برنامجك، وهي تتكون من تلك الكلمة المفتاحية فقط، فإن كانت أدوات المطور مفعَّلة في متصفحك، فسيتوقف البرنامج عند وصوله إلى هذه التعليمة.

8.6 توليد الخطأ

لا يمكن للمبرمج منع كل الأخطاء الوارد حدوثها في البرنامج، خاصةً إذا كان البرنامج يتواصل مع العالم الخارجي بأيّ طريقة كانت، إذ من الممكن تلقيه مدخلات خاطئة، أو تحميله بأحمال ومهام زائدة عن طاقته، أو تفشل الشبكة التي يتواصل من خلالها، وذلك على سبيل المثال لا الحصر.

لا تشغل بالك بشأن تلك المشاكل إذا كنت تبرمج لنفسك فقط، حيث تستطيع تجاهلها إلى حين حدوثها، ثم تتعامل معها حينها، لكن إن كنت تبني برنامجًا أو أداةً لعميل لك، أو برنامجًا سيستخدمه غيرك، فيجب أن تضع في حساباتك احتمالات التصرفات غير السليمة وغير المتوقعة، فقد يكون الحل الأمثل أحيانًا بتجاهل المدخل ليتابع البرنامج العمل، أو تبليغ المستخدم برسالة تفيد ما حدث، لكن يتوجب على البرنامج فعل شيء ما على أساس استجابة للمشكلة الواقعة في أي حالة كانت.

لنقل مثلًا أن لديك دالةً اسمها `promptNumber`، حيث تطلب من المستخدِم إدخال عدد ثم تعيده هي، فلو أدخل المستخدم كلمةً مثل "orange" مثلًا، فما الذي ستعيده هذه الدالة؟

أحد الخيارات المتاحة هي جعل الدالة تعيد قيمةً خاصةً، مثل `null`، أو `undefined`، أو `-1` كما يلي:

```
function promptNumber(question) {
  let result = Number(prompt(question));
  if (Number.isNaN(result)) return null;
```

```
    else return result;
  }

  console.log(promptNumber("How many trees do you see?"));
```

يجب على أيّ شيفرة تستدعي `promptNumber` الآن التحقق من قراءة العدد الفعلي، وإذا فشلت فيجب إصلاح ذلك بطريقة ما، ربما بإعادة السؤال، أو بكتابة قيمة افتراضية، أو بإعادة قيمة خاصة إلى مستدعيها للإشارة إلى فشلها في فعل ما طُلب منها.

وسترى أن مواقف عديدة يصلحها إعادة قيمة خاصة للإشارة إلى خطأ ما، خاصةً في حالة شيوع الخطأ ووجوب وضعه في الحسبان صراحةً؛ لكن هذا له سيئاته، فماذا لو كانت الدالة تستطيع إعادة جميع القيم الممكنة؟ فسيكون عليك فعل شيء ما عند التعامل مع تلك الدالة مثل تغليف النتيجة بكائن لتتمكن من تمييز حالة النجاح من الفشل.

```
function lastElement(array) {
  if (array.length == 0) {
    return {failed: true};
  } else {
    return {element: array[array.length - 1]};
  }
}
```

والمشكلة الثانية عند إعادة قيم خاصة هي أن هذه الإعادة تؤدي إلى شيفرات غريبة، فإن استدعى جزء من الشيفرة الدالة `promptNumber` عشرة مرات، فعليه التحقق من إعادة `null` عشرة مرات أيضًا ، وإن كانت إجابته في التحقق من `null` هي إعادة `null` نفسها، فعلى من يستدعي الدالة التحقق منها بدورها، وهكذا.

8.7 الاستثناءات Exceptions

إذا لم تستطع دالة ما تنفيذ وظيفتها على النحو الذي صممت من أجله، فسيكون الحل هو إيقاف ما نفعله وننتقل فورًا إلى المكان الذي يعرف كيف يعالج هذه المشكلة وذلك العجز، وهذا هو دور معالجة الاستثناءات exception handling.

الاستثناءاتexceptions -وتترجم أحيانًا إلى اعتراضات- ما هي إلا آليات تمكّن الشيفرة التي تواجه مشاكل من رفع استثناء أو تبلغ به، وقد يكون الاستثناء أي قيمة، ويمكن تشبيه هذا البلاغ أو الرفع بإعادة مشحونة نوعًا ما من الدالة، حيث تقفز من الدالة الحالية وممن استدعاها أيضًا لتصل إلى الاستدعاء الأول الذي بدأ التنفيذ الحالي، ويسمى هذا فك المكدس unwinding the stack، ولعلك تذكر مكدس استدعاءات الدالة الذي ذكرناه في الفصل الثالث من هذا الكتاب، إذ يصغِّر الاستثناء هذا المكدس، ملقيًا لكل سياقات الاستدعاء التي يقابلها.

لن تكون الاستثناءات ذات فائدة إن ذهبت مباشرةً إلى قاع المكدس، وما زادت على أن أتت بطريقة جديدة لبعثرة البرنامج، وإنما تظهر قوتها حين تضع عقبات obstacles لالتقاط هذه الاستثناءات وهي ماضية في المكدس، فبمجرد التقاطك لاستثناء ما، فستستطيع التعامل معه ومعالجته لرؤية أصل المشكلة، ثم تتابع تشغيل البرنامج، انظر مثلًا كما يلي:

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
  if (promptDirection("Which way?") == "L") {
    return "a house";
  } else {
    return "two angry bears";
  }
}

try {
  console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

تُستخدم كلمة throw المفتاحية لرفع الاستثناءات، وتُلتقط بتغليف جزء من الشيفرة في كتلة try، متبوعةً بكلمة catch المفتاحية، وحين تتسبب الشيفرة التي في كتلة try في رفع استثناء، فستُقيَّم كتلة catch مع ربط الاسم الموجود بين أقواس بقيمة الاستثناء، وإذا انتهت كتلة catch، أو try دون مشاكل، فسيتابع البرنامج سيره أسفل تعليمة try/catch بكاملها.

استخدمنا في هذه الحالة بانيError لإنشاء قيمة الاستثناء الخاصة بنا، وهو باني جافاسكربت قياسي ينشئ كائنًا مع خاصية message، كما تَجمع نُسَخ هذا الباني في أغلب بيئات جافاسكربت معلومات عن مكدس الاستدعاء الذي كان موجودًا عند إنشاء الاستثناء فيما يسمى بتعقب المكدس stack trace، وتخزن هذه المعلومات في خاصية stack، كما يمكن الاستفادة منها حين محاولة تصحيح مشكلة ما، إذ تخبرنا بالدالة التي حدثت فيها المشكلة وأي الدوال نفّذت هذه الاستدعاء الفاشل.

تتجاهل دالة `look` احتمال أن `promptDirection` قد تخطئ، وهذه مزية كبيرة للاستثناءات، إذ لا تكون شيفرة معالجة الخطأ ضروريةً إلا عند النقطة التي يحدث فيها الخطأ وعند النقطة التي يعالَج فيها؛ أما الدوال التي بين ذلك فلا تكاد تكون مهمةً.

8.8 التنظيف وراء الاستثناءات

يُعَدّ تأثير الاستثناء نوعًا آخرًا من تدفق التحكم، فكل حدث يسبب استثناء -وهو كل استدعاء دالة تقريبًا وكل وصول لخاصية- قد يجعل التحكم يترك شيفرتك فجأة.

فإن كانت الشيفرة بها عدة آثار جانبية، فقد يمنع استثناء ما بعض تلك الآثار من الحدوث، حتى لو كان تدفق التحكم المنتظم لها يشير إلى احتمال حدوثها كلها، فمثلًا، انظر إلى المثال التالي لشيفرة مصرفية سيئة.

```
const accounts = {
  a: 100,
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Enter an account name");
  if (!accounts.hasOwnProperty(accountName)) {
    throw new Error(`No such account: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

تحوِّل دالة `transfer` مبلغًا من المال من حساب ما إلى حساب آخر، مع طلب اسم الحساب الآخر أثناء التحويل، وإذا أُدخل اسم حساب غير صالح، فسترفع `getAccount` استثناءً.

تنقل `transfer` المال أولًا من الحساب الأول، ثم تستدعي `getAccount` قبل إضافة المال إلى حساب جديد، فإن توقف سير عملها بسبب رفع استثناء، فسيختفي المال ويضيع بين الحسابين!

يمكن كتابة تلك الشيفرة بأسلوب أذكى من ذلك من خلال استدعاء `getAccount` قبل البدء بنقل المال مثلًا، لكن للأسف فقد تحدث المشاكل المشابهة لهذه المشكلة بطريقة غير واضحة، كما تكون أقل أثرًا من أن تُلاحظ بمجرد النظر، فحتى الدوال التي لا يُتوقع منها رفع استثناءات، قد ترفعها في ظروف استثنائية أو حين تحتوي على خطأ المبرمج.

إحدى الطرق التي يمكن معالجة ذلك بها، هي التقليل من استخدام الآثار الجانبية، باستخدام أسلوب برمجة يحسب القيم الجديدة بدلًا من تغيير البيانات الموجودة فعليًا، على سبيل المثال لا الحصر، فإذا توقف جزء من الشيفرة عن العمل أثناء إنشاء قيمة جديدة، فلن يرى أحد هذه القيمة غير الجاهزة، ولن نرى مشكلةً من الأساس.

لكن هذا قد لا يكون عمليًا في كل مرة، لذا سننظر في ميزة أخرى في تعليمة `try`، إذ يمكن أن تُتبَع بكتلة `finally` بدلًا من كتلة `catch` أو بالإضافة إليها، وتقول كتلة `finally` "شغِّل هذه الشيفرة مهما حدث، وذلك بعد محاولة تشغيل الشيفرة في كتلة `try`". انظر كما يلي:

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
    accounts[from] -= amount;
    progress = 1;
    accounts[getAccount()] += amount;
    progress = 2;
  } finally {
    if (progress == 1) {
      accounts[from] += amount;
    }
  }
}
```

تتتبع هذه النسخة من الدالة مدى تقدمها وسيرها، وإذا تسبب في حالة غير مستقرة للبرنامج عند خروجها، فإنها تصلح أثر هذا الخلل الذي أحدثته.

لاحظ أن شيفرة `finally` رغم تشغيلها عند رفع استثناء في كتلة `try`، إلا أنها لا تتدخل في الاستثناء نفسه، وعليه فإن المكدس سيستمر في تفكيك نفسه بعد تشغيل كتلة `finally`.

كتابة مثل هذه البرامج التي تعمل بكفاءة ويعتمد عليها حتى في حالات ظهور استثناءات في أماكن غير متوقعة أمر صعب وليس سهلًا.

لا يبالي الكثير من الناس بهذا، فقد لا تحدث المشكلة إلا نادرًا جدًا بحيث لا يمكن ملاحظتها، وذلك بسبب عدم ظهور الاستثناءات إلا في الحالات الاستثنائية، وإذا أردنا القول بأن هذا شيء جيد أو سيء جدًا، فيجب النظر أولًا إلى الأثر والتخريب الذي يحدث عند فشل البرنامج أو تعطله.

8.9 الالتقاط الانتقائي

تعالج البيئة الاستثناء الذي يمر في المكدس كله دون التقاطه، ويختلف هنا ما يحدث باختلاف البيئة نفسها، ففي المتصفحات مثلًا، يُكتب وصف الخطأ إلى طرفية جافاسكربت والتي يمكن الوصول إليها من خلال أدوات المتصفح أو قائمة المطورين Developers Menu؛ أما في Node.js فستكون بيئة جافاسكربت الغير موجودة في متصفح والتي سنناقشها في الفصل العشرين، أكثر حذرًا بشأن تدمير البيانات، فتُخرِج العملية كلها عند حدوث استثناء غير معالَج unhandled.

بالنسبة لأخطاء المبرمجين، فأفضل شيء يمكن فعله هو ترك الخطأ يمر بسلام دون لمسه، كما يمثل الاستثناء الغير معالَج طريقةً معقولةً ومنطقيةً للإشارة إلى وجود عطل في البرنامج، وستعطيك طرفية جافاسكربت في المتصفحات الحديثة بعض المعلومات عن استدعاءات الدالة التي كانت في المكدس حين حدوث المشكلة؛ أما بالنسبة للمشاكل المتوقع حدوثها أثناء الاستخدام العادي، فسيُنبِئ توقف البرنامج بسبب استثناء غير معالَج استراتيجيةً سيئةً جدًا.

كما تتسبب الاستخدامات غير الصحيحة للغة مثل الإشارة المرجعية لرابطة غير موجودة، أو البحث عن خاصية في `null`، أو استدعاء شيء غير الدوال، في رفع استثناءات، كما يمكن التقاط تلك الاستثناءات.

كل ما نستطيع معرفته عند دخول متن `catch` هو أن شيئًا ما داخل متن `try` قد تسبب في رفع استثناء، لكن لا نستطيع معرفة ماهية الاستثناء نفسه أو ما فعله.

لا توفر جافاسكربت -في إغفال صارخ- دعمًا مباشرًا لاستثناءات الالتقاط الانتقائي، فإما تلتقطها كلها أو لا تدرك منها شيئًا، وقد يحلو للمرء افتراض أن الاستثناء الذي حصل عليه هو الذي كان يفكر فيه ويتوقعه حين كتب كتلة `catch`، بسبب هذا الخطأ في اللغة، لكن سيخبرك الواقع أنه قد لا يحدث هذا دومًا معك، فلعله تم اختراق افتراض آخر، أو لعلك تسببت في زلة أحدثت استثناء؛ ويحاول المثال التالي استدعاء `promptDirection` إلى أن يحصل على إجابة صالحة:

```
for (;;) {
  try {
    let dir = promtDirection("Where?"); // ← typo!
    console.log("You chose ", dir);
    break;
  } catch (e) {
```

```
    console.log("Not a valid direction. Try again.");
  }
}
```

تُعَدّ بُنية `for(;;)` طريقةً متعمَّدة لإنشاء حلقة تكرارية لا تنهي نفسها، ولا نخرج منها إلا حين حصولنا على اتجاه صالح، لكننا أخطأنا في تهجئة `promptDirection` التي أعطتنا الخطأ "متغير غير معرَّف" undefined variable، وتتعامل كتلة `catch` تعاملًا خاطئًا مع خطأ الرابطة على أنه مؤشر إدخال غير صالح، وذلك بسبب تجاهلها قيمة اعترضها (`e`) مفترضةً أنها تعرف المشكلة، كما لا يتسبب هذا في حلقة لا نهائية فحسب، بل يدفن رسالة الخطأ المفيدة التي نريدها عن الرابطة التي أُخطئ في هجائها.

تقول القاعدة العامة لا تَلتقط الاستثناءات التقاطًا كليًا إلا إذا كان بغرض توجيهها إلى مكان ما مثل توجيهها عبر الشبكة مثلًا لإخبار نظام آخر بتعطل برنامجنا، وعليك التفكير مليًا حتى حينئذ حول كيفية إخفاء المعلومات، وعلى ذلك فإننا نريد التقاط نوع محدد من الاستثناءات من خلال التحقق داخل كتلة `catch` مما إذا كان الاستثناء الذي حصلنا عليه هو الذي نريده أم لا، وإن لم يكن فنعيد رفعه، لكن كيف نعرف الاستثناء الذي نريده أصلًا؟

نستطيع موازنة خاصية `message` التابعة له برسالة الخطأ التي نتوقعها، لكن ليست هذه هي الطريقة المثلى للبرمجة، ففعلنا هذا ما هو إلا استخدام لمعلومات مخصصة لاطلاع العنصر البشري عليها أي الرسالة، وذلك لبناء قرار برمجي على هذه المعلومات، فإذا غير أحد هذه الرسالة أو ترجمها، فستتعطل الشيفرة مرةً أخرى، والحل البديل هو تعريف نوع جديد من الأخطاء واستخدام `instanceof` لتعريفه:

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

يوسع صنف الخطأ الجديد `Error`، حيث لا يعرّف بانيًا خاصًا به، مما يعني أنه يرث باني `Error` الذي يتوقع رسالة نصية على أساس وسيط، كما لا يعرِّف أي شيء أصلًا، وهو -أي الصنف- فارغ.

تتصرف كائنات `InputError` مثل كائنات `Error` باستثناء امتلاكها لصنف مختلف يمكننا تمييزها به، وتستطيع الحلقة التكرارية الآن التقاط هؤلاء بطريقة أكثر حذرًا:

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Not a valid direction. Try again.");
    } else {
      throw e;
    }
  }
}
```

وهذا سيلتقط نُسخًا من `InputError` فقط وتترك الاستثناءات غير المرتبطة به تمر، فإذا أعدت إدخال خطأ الهجاء مرةً أخرى، فسيبَلَّغ عن خطأ رابطة غير معرَّفة هذه المرة.

8.10 التوكيدات Assertions

التوكيدات هي عمليات تحقق داخل البرنامج، حيث تنظر هل الشيء موجود على الصورة التي يفترض به أن يكون عليها أم لا، وتُستخدَم للبحث عن أخطاء المبرمجين، وليس لمعالجة مواقف يمكن حدوثها في التشغيل العادي، فمثلًا، إذا وُصف `firstElement` على أساس دالة لا يمكن استدعاؤها على مصفوفة فارغة، فربما نكتبها كما يلي:

```
function firstElement(array) {
  if (array.length == 0) {
    throw new Error("firstElement called with []");
  }
  return array[0];
}
```

ستبعثِر الشيفرة السابقة برنامجك بمجرد إساءة استخدامها، بدلًا من إعادة `undefined` بصمت مثل التي تحصل عليها حين تقرأ خاصية مصفوفة غير موجودة، ويجعل ذلك من الصعب لمثل تلك الأخطاء المرور دون ملاحظتها من أحد، وأسهل في معرفة سببها عند وقوعها.

لكن هذا لا يعني أننا ننصح بكتابة توكيدات لكل نوع من المدخلات الخاطئة، فهذا عمل كثير جدًا، كما سينشئ لنا شيفرة غير صافية وملأى بأكواد غير ضرورية، بل احفظ هذه التوكيدات من أجل أخطاء يسهل ارتكابها، أو أخطاء تقع فيها بنفسك.

8.11 خاتمة

المدخلات الخاطئة والأخطاء عمومًا هي أشياء لازمة لنا في الحياة، ومن المهم في البرمجة العثور عليها، وتشخيصها، وإصلاحها، حيث ستظهر هنا في البرمجة على صورة زلات برمجية bugs. وقد تصير المشاكل أسهل في ملاحظتها إذا كان لديك حزمة اختبار مؤتمتة، أو إذا أضفت توكيدات إلى برامجك.

يجب معالجة المشاكل التي تحدث بسبب عوامل خارجة عن تحكم البرنامج بلطف وحكمة، فقد تكون القيم الخاصة المعادة هي الطريقة المثلى لتتبع هذه المشاكل ومعالجتها، وإلا فربما نود استخدام الاستثناءات.

سيتسبب رفع الاستثناءات في فك مكدس الاستدعاء حتى يصل إلى كتلة `try/catch` أو إلى نهاية المكدس، وستُعطَى قيمة الاستثناء إلى كتلة `catch` التي تلتقطها، مما يؤكد لنا أن هذا هو نوع الاستثناء الذي نريده قبل إجراء أي فعل عليه، كما يمكن استخدام كتل `finally` للمساعدة في تحديد تدفقات التحكم غير المتوقعة التي تحدث بسبب الاستثناءات، وذلك من أجل التأكد من تشغيل جزء من الشيفرة بعد انتهاء الكتلة.

8.12 تدريبات

8.12.1 Retry

لنقل أنه لديك دالة `primitiveMultiply` التي تضرب عددين معًا في 20 بالمئة من الحالات، وترفع استثناءًا في الثمانين بالمئة الباقية من نوع `MultiplicatorUnitFailure`.

اكتب دالة تغلف هذه الدالة وتظل تحاول حتى نجاح أحد الاستدعاءات، كما تعيد النتيجة بعد ذلك، وتأكد من معالجة الاستثناءات التي تريد معالجتها فقط.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
class MultiplicatorUnitFailure extends Error {}

function primitiveMultiply(a, b) {
  if (Math.random() < 0.2) {
    return a * b;
  } else {
    throw new MultiplicatorUnitFailure("Klunk");
```

```
  }
}

function reliableMultiply(a, b) {
  // شيفرتك هنا.
}

console.log(reliableMultiply(8, 8));
// → 64
```

إرشادات الحل

يجب حدوث استدعاء `primitiveMultiply` داخل كتلة `try` قطعًا، ويجب على كتلة `catch` الموافقة لهذا رفع استثناء مرةً أخرى إذا لم تكن نسخةً من `MultiplicatorUnitFailure`، كما تتأكد من إعادة محاولة الاستدعاء مرةً أخرى حين تكون نسخةً منها.

استخدِم حلقةً تكراريةً لتكرار المحاولة، بحيث لا تتوقف إلا عند نجاح الاستدعاء، كما في مثال `look` الذي ذكرناه آنفًا في هذا الفصل، أو استخدِم التعاودية recursion على أمل عدم الحصول على سلسلة نصية من الفشل لفترة طويلة بحيث تؤدي إلى طفحان المكدس، وهذا هو الخيار الآمن بالمناسبة.

8.12.2 الصندوق المغلق

انظر الكائن التالي:

```
const box = {
  locked: true,
  unlock() { this.locked = false; },
  lock() { this.locked = true;  },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this._content;
  }
};
```

ما هذا إلا صندوق به قفل، وهناك مصفوفة داخل الصندوق، حيث لا تستطيع الوصول إليها إلا حين يُفتح الصندوق، وأنت ممنوع من الوصول إلى خاصية `_content` الخاصة مباشرةً.

اكتب دالة اسمها `withBoxUnlocked` تأخذ قيمة دالة على أساس وسيط، وتفتح الصندوق، كما تشغِّل الدالة، ثم تتأكد أن الصندوق مقفل مرةً أخرى قبل الإعادة، بغض النظر عن إعادة الدالة الوسيطة بطريقة طبيعية أو رفع استثناء.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
const box = {
  locked: true,
  unlock() { this.locked = false; },
  lock() { this.locked = true;  },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this._content;
  }
};

function withBoxUnlocked(body) {
  // شيفرتك هنا.
}

withBoxUnlocked(function() {
  box.content.push("gold piece");
});

try {
  withBoxUnlocked(function() {
    throw new Error("Pirates on the horizon! Abort!");
  });
} catch (e) {
  console.log("Error raised: " + e);
}
console.log(box.locked);
// → true
```

للمزيد من النقاط، حين يكون الصندوق مغلقًا، تأكد من بقائه مغلقًا إذا استدعيت `withBoxUlocked`.

إرشادات الحل

يستدعي هذا التدريب كتلة `finally`، ويجب على الدالة فتح الصندوق أولًا، ثم تستدعي الدالة الوسيطة من داخل متن كتلة `try`، وبعدها تقوم كتلة `finally` التي تليها بغلق الصندوق.

للتأكد من عدم إغلاق الصندوق إذا لم يكن مغلقًا من البداية، تحقق من إغلاقه عند بدء الدالة، ولا تفتحه وتغلقه إلا إذا كان مغلقًا في البداية.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

9. التعابير النمطية Regular Expressions

يبادر البعض إذا قابلَته مشكلة إلى استخدام التعابير النمطية، ولا يدرك المرء أنه بهذا قد جعل المشكلة اثنتين.

— جيمي زاوينيسكي Jamie Zawiniski

إذا قطعت الخشب في عكس اتجاه أليافه فستحتاج إلى قوة أكبر، كذلك إذا سار البرنامج عكس اتجاه المشكلة، إذ ستحتاج شيفرات أكثر لحلها.

— يوان-ما Yuan-Ma، كتاب البرمجة The Book of Programming.

إذا نظرنا إلى التقنيات والأدوات البرمجية المنتشرة، فسنرى أنّ التقنية أو الأداة المشهورة هي التي تُثبت كفاءتها العملية في مجالها، أو التي تتكامل تكاملًا ممتازًا مع تقنية أخرى مستخدَمة بكثرة، وليس لأنها الأجمل أو الأذكى.

سنناقش هنا إحدى تلك الأدوات التي تدعى بالتعابير النمطية Regular Expressions، وهي طريقة لوصف الأنماط patterns في بيانات السلاسل النصية، إذ تُكوّن هذه التعابير لغةً صغيرةً مستقلةً بذاتها، لكنها رغم ذلك تدخل في لغات برمجية أخرى مثل جافاسكربت، كما تدخل في العديد من الأنظمة.

قد تكون صيغة التعابير النمطية غريبةً للناظر إليها، كما أنّ الواجهة التي توفرها جافاسكربت للتعامل معها ليست بالمثلى، لكن رغم هذا، فتلك التعابير تُعَد أداةً قويةً لفحص ومعالجة السلاسل النصية، كما سيعينك فهمها على كتابة حلول أكفأ للمشاكل التي تواجهك.

9.1 إنشاء تعبير نمطي

التعبير النمطي هو نوع من أنواع الكائنات؛ فإما يُنشأ باستخدام الباني RegExp، أو يُكتب على أساس قيمة مصنَّفة النوع literal value بتغليف النمط بشرطتين مائلتين أماميتين /.

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

يُمثِّل كائنا التعبيرات النمطية في المثال السابق النمط نفسه أي محرف a يتبعه b ثم c، ويُكتب النمط على أساس سلسلة نصية عادية عند استخدام الباني RegExp، لذا تطبَّق القواعد المعتادة للشرطات المائلة الخلفية \، على عكس الحالة الثانية التي نرى فيها النمط بين شرطتين مائلتين، إذ تعامَل الشرطات هنا تعاملًا مختلفًا.

نحتاج إلى وضع شرطة خلفية قبل أي شرطة أمامية نريدها جزءًا من النمط نفسه، وذلك لأن الشرطة الأمامية تنهي النمط إذا وُجدت. كذلك فإن الشرطات الخلفية التي لا تكون جزءًا من ترميز خاص لمحرف مثل \n لن تُتَجاهل كما نفعل في السلاسل النصية، وعليه ستتسبب في تغيير معنى النمط.

تملك بعض المحارف مثل علامات الاستفهام وإشارات الجمع مَعاني خاصة في التعابير النمطية، حيث سنحتاج إلى وضع شرطة مائلة خلفية قبلها إذا أردنا لها تمثيل المحرف نفسه وليس معناه في التعابير النمطية.

```
let eighteenPlus = /eighteen\+/;
```

9.2 التحقق من المطابقات

تملك كائنات التعابير النمطية توابع عديدة، وأبسط تلك التوابع هو التابع test الذي إن مرَّرنا سلسلةً نصيةً إليه، فسيُعيد قيمةً بوليانيةً تخبرنا هل تحتوي السلسلة على تطابق للنمط الذي في التعبير أم لا.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

يتكون التعبير النمطي من محارف عادية غير خاصة تمثِّل -ببساطة- ذلك التسلسل من المحارف، فإذا وُجد abc في أيّ مكان في السلسلة النصية التي نختبرها -ولا يُشترط وجودها في بدايتها-، فسيُعيد test القيمة true.

9.3 مجموعات المحارف

يمكن التحقق من وجود abc في سلسلة نصية باستدعاء التابع indexof.

تسمح لنا التعابير النمطية بالتعبير عن أنماط أكثر تعقيدًا، فمثلًا كل ما علينا فعله لمطابقة عدد ما هو وضع مجموعة من المحارف بين قوسين مربعين لجعل ذلك الجزء من التعبير يطابق أيًا من المحارف الموجودة بين الأقواس.

لننظر المثال التالي حيث يطابق التعبيران جميع السلاسل النصية التي تحتوي على رقم ما:

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

يمكن الإشارة إلى مجال من المحارف داخل القوسين المربعين باستخدام الشرطة - بين أول محرف فيه وآخر محرف، ويُحدَّد الترتيب في تلك المحارف برمز اليونيكود لكل محرف -كما ترى من المثال أعلاه الذي يشير إلى مجال الأرقام من 0 إلى 9-، وهذه الأرقام تحمل رمز 48 حتى 57 بالترتيب في اليونيكود، وعليه فإنّ المجال `[0-9]` يشملها جميعًا، ويطابق أي رقم.

تمتلك مجموعة محارف الأعداد اختصارات خاصة بها كما هو شأن العديد من مجموعات المحارف الشائعة، فإذا أردنا الإشارة إلى المجال من 0 حتى 9، فسنستخدِم الاختصار `\d`.

الاختصار	الدلالة
`\d`	أيّ محرف رقمي
`\w`	محرف أبجدي أو رقمي، أي محرف الكلمة
`\s`	أيّ محرف مسافة بيضاء، مثل المسافات الفارغة والجداول والأسطر الجديدة وما شابهها
`\D`	محرف غير رقمي
`\W`	محرف غير أبجدي وغير رقمي
`\S`	محرف مغاير للمسافة البيضاء
`.`	أيّ محرف عدا السطر الجديد

نستطيع مطابقة تنسيق التاريخ والوقت -كما في 01-30-2003 15:20- باستخدام التعبير التالي:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("01-30-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

تعيق الشرطات المائلة الموجودة في المثال أعلاه قراءة النمط الذي نعبر عنه، وسنرى لاحقًا نسخةً أفضل في هذا الفصل.

يمكن استخدام رموز الشرطات المائلة تلك داخل أقواس مربعة، إذ تعني `[\d.]` مثلًا أيّ رقم أو محرف النقطة `.` لكن تفقِد النقطة نفسها معناها المميز لها إذا كانت داخل أقواس مربعة، وبالمثل في حالة إشارة

الجمع +؛ أما إذا أردنا عكس مجموعة محارف، أي إذا أردنا التعبير عن رغبتنا في مطابقة أيّ محرف عدا تلك المحارف التي في المجموعة، فنستخدم رمز الإقحام ^ بعد قوس الافتتاح.

```
let notBinary = /[^01]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

9.4 تكرار أجزاء من النمط

لقد بتنا الآن نعلم كيف نطابق رقمًا واحدًا، لكن كيف سنفعل ذلك إذا أردنا مطابقة عدد يحتوي على أكثر من رقم؟ إذا وضعنا إشارة الجمع + بعد شيء ما في تعبير نمطي، فستشير إلى أنّ هذا العنصر قد يكرَّر أكثر من مرة، بالتالي تعني /\d+/ مطابقة محرف رقم أو أكثر.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

تحمل إشارة النجمة * معنى قريبًا من ذلك، حيث تسمح للنمط بالمطابقة على أي حال، فإذا لحقت إشارة النجمة بشيء ما، فلن تمنع النمط من مطابقته، إذ ستطابق نُسخًا صفرية zero instances حتى لو لم تجد نصًا مناسبًا لمطابقته؛ أما إذا جاءت علامة الاستفهام بعد محرف في نمط، فستجعل ذلك المحرف اختياريًا optional، أي قد يحدث مرةً واحدةً أو لا يحدث.

يُسمح للمحرف u في المثال التالي بالحدوث، ويحقق النمط المطابقة حتى لو لم يكن u موجودًا أيضًا.

```
let neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

نستخدم الأقواس المعقوصة إذا أردنا حدوث النمط عددًا معينًا من المرات، فإذا وضعنا `{4}` بعد عنصر ما مثلًا، فسيجبره بالحدوث 4 مرات حصرًا، ومن الممكن تحديد المجال الذي يمكن للعنصر حدوثه فيه بكتابة `{2,4}` التي تشير إلى وجوب ظهور العنصر مرتين على الأقل، وأربع مرات على الأكثر.

لدينا نسخة أخرى من نمط التاريخ والوقت، حيث تسمح بذكر الأيام برقم واحد -أو رقمين-، والأشهر، والساعات، إذ تُعَدّ أسهل قليلًا في قراءتها، وهي المثال الذي قلنا أننا سنعود إليه بنسخة أفضل.

```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("1-30-2003 8:45"));
// → true
```

نستطيع تحديد مجالات مفتوحة عند استخدام الأقواس بإهمال الرقم الموجود بعد الفاصلة، وبالتالي، تعني `{5,}` خمس مرات على الأقل.

9.5 جمع التعبيرات الفرعية

إذا أردنا استخدام عامل مثل `*`، أو `+` على أكثر من عنصر في المرة الواحدة، فيجب استخدام الأقواس، وسترى العوامل الجزء الذي داخل الأقواس من التعبير النمطي عنصرًا واحدًا.

```
let cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

تطبَّق إشارتا الجمع الأولى والثانية على o الثانية فقط في boo، وhoo على الترتيب؛ أما علامة الجمع الثالثة فتطبَّق على المجموعة كلها `(hoo+)` مطابِقةً تسلسلًا واحدًا أو أكثر بهذا.

يجعل محرف `i` -الذي في نهاية التعبير- هذا التعبير النمطي غير حساس لحالة المحارف، إذ يسمح بمطابقة المحرف B في سلسلة الدخل النصية رغم تكوّن النمط من محارف صغيرة.

9.6 التطابقات والمجموعات

يُعَدّ التابع `test` أبسط طريقة لمطابقة تعبير نمطي، إذ لا يخبرك إلا بمطابقة التعبير النمطي من عدمها وفقط، كذلك تملك التعبيرات النمطية تابعًا اسمه `exec`، حيث يُعيد القيمة `null` إذا لم يجد مطابقة، كما يُعيد كائنًا مع معلومات عن المطابقة إذا وجد تطابق.

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
```

```
console.log(match.index);
// → 8
```

تكون للكائن المعاد من exec خاصية تدعى index، إذ تخبرنا أين تبدأ المطابقة الناجحة للسلسلة النصية؛ أما خلاف هذا فيبدو الكائن أشبه بمصفوفة من السلاسل النصية -وهو كذلك حقًا-، ويكون أول عنصر في تلك المصفوفة هو السلسلة المطابَقة، كما يكون ذلك هو تسلسل الأرقام الذي كنا نبحث عنه في المثال السابق.

تحتوي قيم السلاسل النصية على التابع match الذي له سلوك مشابه:

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

حين يحتوي التعبير النمطي على تعبيرات فرعية مجمَّعة داخل أقواس، فسيظهر النص الذي يطابق تلك المجموعات في مصفوفة، ويكون العنصر الأول هو التطابق كله دومًا، في حين يكون العنصر التالي هو الجزء المطابَق بواسطة المجموعة الأولى التي يأتي قوس افتتاحها أولًا في التعبير، ثم المجموعة الثانية، وهكذا.

```
let quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

إذا لم تطابَق مجموعة ما مطلقًا -كأن تُتبَع بعلامة استفهام-، فسيكون موضعها في مصفوفة الخرج غير معرَّفًا undefined، وبالمثل، فإذا طابقت مجموعةً ما أكثر من مرة، فستكون المطابقة الأخيرة هي التي في المصفوفة فقط.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

تفيدنا المجموعات في استخراج أجزاء من سلسلة نصية، فإذا أردنا التحقق من احتواء السلسلة النصية على تاريخ، ومن ثم استخراج ذلك التاريخ وبناء كائن يمثله؛ فيمكننا إحاطة الأنماط الرقمية بأقواس، وأخذ التاريخ مباشرةً من نتيجة exec، لكن نحتاج قبل ذلك إلى النظر سريعًا على الطريقة المضمَّنة لتمثيل قيم التاريخ والوقت في جافاسكربت.

9.7 صنف التاريخ

تحتوي جافاسكربت على صنف قياسي لتمثيل البيانات -أو النقاط- في الزمن، ويسمى ذلك الصنف Date، فإذا أنشأنا كائن تاريخ باستخدام new، فسنحصل على التاريخ والوقت الحاليين.

```
console.log(new Date());
// → Mon Nov 13 2017 16:19:11 GMT+0100 (CET)
```

من الممكن إنشاء كائن لوقت محدد:

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

تستخدِم جافاسكربت تقليدًا تبدأ فيه أعداد الشهور بالصفر -وعليه يكون شهر ديسمبر هو عدد 11-، بينما تبدأ أرقام الأيام بالواحد، وذلك أمر محيِّر وسخيف كذلك لكنه واقع، وإنما ذكرناه للتنبيه.

تُعَدّ آخر أربعة وسائط arguments -أي الساعات والدقائق والثواني والميلي ثانية- وسائط اختيارية، وإذا لم تحدد قيمة أيّ منهم فتكون صفرًا افتراضيًا.

تُخزَّن العلامات الزمنية Timestamps بعدد من الميلي ثانية منذ العام 1970 في منطقة UTC الزمنية -أي التوقيت العالمي-، ويتبع هذا اصطلاحًا ضُبِط بواسطة توقيت يونكس Unix time الذي اختُرع في تلك الفترة أيضًا، كما يمكن استخدام الأرقام السالبة للتعبير عن الأعوام التي سبقت 1970.

إذا استُخدم التابع `getTime` على كائن تاريخ، فسيُعيد ذلك العدد، وهو عدد كبير كما هو متوقع.

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

إذا أعطينا الباني `Date` وسيطًا واحدًا، فسيعامَل الوسيط على أنه تعداد الميلي ثانية، ويمكن الحصول على القيمة الحالية لتعداد المللي ثانية بإنشاء كائن `Date` جديد، واستدعاء `getTime` عليه، أو استدعاء الدالة `Date.now`.

توفِّر كائنات التاريخ توابعًا، مثل:

- `getFullYear`
- `getMonth`
- `getDate`
- `getHours`
- `getMinutes`

- getSeconds

من أجل استخراج مكوناتها، كما يعطينا التابع getYear السنة بعد طرح 1900 منها، لكن لن نستخدِمه كثيرًا لعدم وجود فائدة حقيقية منه. نستطيع الآن إنشاء كائن تاريخ من سلسلة نصية بما أننا وضعنا أقواسًا حول أجزاء التعبير التي تهمنا، أي كما يلي:

```
function getDate(string) {
  let [_, month, day, year] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(year, month - 1, day);
}
console.log(getDate("1-30-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

تُهمَل رابطة الشرطة السفلية _، ولا تُستخدَم إلا لتجاوز عنصر المطابَقة التامة في المصفوفة التي يُعيدها التابع exec.

9.8 حدود الكلمة والسلسلة النصية

يستخرِج التابع getDate التاريخ 00-1-3000 من السلسلة النصية "100-1-30000"، وهو تاريخ غير منطقي لا شك، حيث تحدث المطابقة في أي موقع في السلسلة النصية، لذا تبدأ في حالتنا عند المحرف الثاني، وتنتهي عند المحرف الثاني من النهاية.

نضيف العلامتين ^ و$ لإجبار المطابقة على النظر في السلسلة كلها، إذ تطابِق علامة الإقحام بداية سلسلة الدخل، بينما تطابِق علامة الدولار نهايتها، إذ تطابق /^\d+$/ مثلًا سلسلةً مكونةً من رقم واحد أو أكثر، وتطابق /^!/ أي سلسلة تبدأ بعلامة تعجب، بينما لا تطابق /x^/ أي سلسلة نصية، إذ لا يمكن وجود x قبل بداية السلسلة. نستخدم العلامة \b للتأكد من أنّ التاريخ يبدأ وينتهي عند حدود كلمة، وقد يكون حد الكلمة بداية السلسلة، أو نهايتها، أو أي نقطة فيها تملك محرف كلمة -كما في \w على أحد الجانبين، و محرف غير كلمي على الجانب الآخر.

```
console.log(/cat/.test("concatenate"));
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

لاحظ أنّ علامة الحد لا تطابق محرفًا حقيقيًا، بل تضمن عدم مطابقة التعبير النمطي إلا عند حدوث حالة معينة في الموضع الذي يظهر فيه في النمط.

9.9 أنماط الاختيار

لنقل أننا نريد معرفة هل يحتوي جزء ما من النص على عدد متبوع بإحدى الكلمات التالية: horse، أو cow، أو chicken، أو أي صورة من صور الجمع لها، قد نكتب ثلاثة تعابير نمطية ونختبرها لكن ثَم طريقة أفضل، وذلك بوضع محرف الأنبوب | الذي يشير إلى خيار بين النمط الذي عن يمينه والنمط الذي عن يساره، وعليه نستطيع القول كما يلي:

```
let animalCount = /\b\d+ (horse|cow|chicken)s?\b/;
console.log(animalCount.test("15 horses"));
// → true
console.log(animalCount.test("15 horsechickens"));
// → false
```

يمكن استخدام الأقواس لتقييد جزء النمط الذي يطبَّق عليه عامل الأنبوب، ويمكن وضع عدة عوامل مثل هذا بجانب بعضها البعض للتعبير عن اختيار بين أكثر من بديلين اثنين.

9.10 آلية المطابقة

يبحث محرك التعبير نظريًا عند استخدام `exec` أو `test` عن تطابق في سلسلتنا النصية، وذلك بمحاولة مطابقة التعبير من بداية السلسلة أولًا، ثم من المحرف الثاني، وهكذا حتى يجد تطابقًا أو يصل إلى نهاية السلسلة، وعندئذ يُعيد أول تطابق وجده، أو يكون قد فشل في إيجاد تطابق أصلًا؛ أما في عملية المطابقة الفعلية، فيعامِل المحرك التعبير النمطي مثل مخطط تدفق flow diagram، وإذا استخدمنا مثالنا السابق عن الحيوانات، فسيبدو مخطط التعبير الخاص بها كما يلي:

يكون تعبيرنا مطابقًا إذا استطعنا إيجاد مسار من جانب المخطط الأيسر إلى جانبه الأيمن، حيث نحفظ الموضع الحالي في السلسلة النصية، ونتأكد في كل حركة نتحركها خلال صندوق من أنّ جزء السلسلة التالي لموضعنا الحالي يطابق ذلك الصندوق.

إذا كنا نحاول مطابقة `"the 3 horses"` من الموضع 4، فسيبدو مسار تقدمنا داخل المخطط كما يلي:

- يمكننا تجاوز الصندوق الأول لوجود حد كلمي word boundary عند الموضع 4.
- لا زلنا في الموضع 4 ووجدنا رقمًا، لذا نستطيع تجاوز الصندوق الثاني.
- يتكرر أحد المسارات إلى ما قبل الصندوق الثاني (الرقم) عند الموضع 5، بينما يتحرك الآخر للأمام خلال الصندوق، ويحمل محرف مسافة واحد؛ لذا يجب أخذ المسار الثاني لوجود مسافة وليس رقمًا.
- نحن الآن في الموضع 6، أي بداية horses وعند التفرع الثلاثي في المخطط، إذ لا نرى cow، ولا chicken هنا، لكن نرى horse، لذا سنأخذ ذلك الفرع.
- يتخطى أحد المسارات صندوق s عند الموضع 9 بعد التفرع الثلاثي، ويذهب مباشرةً إلى حد الكلمة الأخير، بينما يطابق المسار الآخر s، كما سنمر خلال صندوق s لوجود المحرف s وليس حدًا لكلمة.
- نحن الآن عند الموضع 10 وهو نهاية السلسلة النصية، ونستطيع مطابقة حد الكلمة فقط، وتُحسب نهاية السلسلة النصية على أنها حد كلمي، لذا سنمر خلال الصندوق الأخير ونكون قد طابقنا تلك السلسلة النصية بنجاح.

9.11 التعقب الخلفي Backtracking

يطابق التعبير النمطي `/\b([01]+b|[\da-f]+h|\d+)\b/` عددًا ثنائيًا متبوعًا بـ b، أو عددًا ست عشريًا hexadecimal -وهو نظام رقمي تمثل فيه الأعداد من 10 إلى 15 بالأحرف a حتى f- متبوعًا بـ h، أو عددًا عشريًا عاديًا ليس له محرف لاحق، وفيما يلي المخطط الذي يصف ذلك:

سيدخل الفرع العلوي الثنائي عند مطابقة ذلك التعبير حتى لو لم يحوي الدخل على عدد ثنائي، حيث سيصبح من الواضح عند المحرف 3 مثلًا أننا في الفرع الخاطئ عند مطابقة السلسلة "103"، فعلى الرغم تطابق السلسلة للتعبير، إلا أنها لا تطابق الفرع الذي نحن فيه.

يبدأ هنا المطابِق matcher بالتعقب الخلفي، فيتذكر موضعه الحالي عند دخول فرع ما -وهو بداية السلسلة في هذه الحالة، بعد صندوق "حد الكلمة" الأول في المخطط-، وذلك ليستطيع العودة والنظر في فرع آخر إذا لم ينجح الفرع الحالي.

سيجرب الفرع الخاص بالأرقام الست عشرية في حالة السلسلة النصية "103" إذا وصل إلى المحرف 3، لكنه سيفشل مجددًا لعدم وجود h بعد العدد، وهنا سيحاول في الفرع الخاص بالأعداد العشرية، وتنجح المطابقة، ويُبلَّغ بها.

يتوقف المطابِق عندما يجد مطابقةً تامةً، وهذا يعني أنه حتى لو كان لدينا فروع متعددة يمكنها مطابقة سلسلة نصية ما، فهو لن يُستخدَم إلا الفرع الأول الذي ظهر وفقًا لترتيبه في التعبير النمطي.

ويحدث التعقب الخلفي أيضًا لعوامل التكرار، مثل + و*، فإذا طابقنا /^.*x/ مع "abcxe"، فسيحاول الجزء *. أخذ السلسلة كلها أولًا، لكن سيدرك المحرك أنه يحتاج إلى x كي يطابق النمط، وبما أنه لا توجد x بعد نهاية السلسلة، فسيحاول عامل النجمة المطابقة من غير المحرف الأخير، لكن لا يعثر المطابِق على x بعد abcx، فيعود أدراجه بالتعقب الخلفي ليطابق عامل النجمة مع abc فقط، وهنا يجد x حيث يحتاجها، ويبلِّغ بمطابقة ناجحة من الموضع 0 حتى 4.

قد يحدث ونكتب تعبيرًا نمطيًا ينفذ الكثير من عمليات التعقب الخلفي، وهنا تحدث مشكلة حين يستطيع النمط مطابقة جزء من الدخل بطرق عديدة مختلفة، فإذا لم ننتبه عند كتابة تعبير نمطي لعدد ثنائي، فقد نكتب شيئًا مثل /([01]+)+b/.

عندما يحاول التعبير مطابقة سلسلة طويلة من الأصفار والآحاد التي ليس لها لاحقة b، فسيمر المطابِق على الحلقة الداخلية حتى تنتهي الأرقام، ثم يلاحظ عدم وجود المحرف b، فينفِّذ تعقبًا خلفيًا لموضع واحد فقط، ثم يمر على الحلقة الخارجية مرةً واحدةً قبل أن يستسلم ويعود أدراجه ليتعقب الحلقة الداخلية مرةً أخرى، كما

سيظل يحاول جميع الطرق الممكنة عبر هاتين الحلقتين، وهذا يعني مضاعفة مقدار العمل مع كل محرف إضافي، فلو أضفنا بعض العشرات من المحارف، لاستغرقت عملية المطابقة إلى ما لا نهاية.

9.12 التابع replace

تحتوي قيم السلاسل النصية على التابع `replace` الذي يمكن استخدامه لاستبدال سلسلة نصية بجزء من سلسلة أخرى.

```
console.log("papa".replace("p", "m"));
// → mapa
```

يمكن أن يكون الوسيط الأول تعبيرًا نمطيًا، وعندئذ يُستبدل التطابق الأول للتعبير النمطي؛ أما إذا أضيف الخيار `g` -اختصارًا لـ global- إلى التعبير النمطي، فتُستبدل جميع التطابقات.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

لو كان لدينا وسيط إضافي للتابع `replace` بحيث نختار منه استبدال تطابق واحد أو جميع التطابقات، لكان أفضل من الاعتماد على خاصية للتعبير النمطي، بل لو كان من خلال توفير تابع مختلف باسم `replaceAl` لكان أفضل.

يأتي مكمن القوة في استخدام التعابير النمطية مع التابع `replace` من حقيقة استطاعتنا الإشارة إلى المجموعات المطابَقة في السلسلة النصية البديلة، فمثلًا، لدينا سلسلة كبيرة تحتوي على أسماء أشخاص، بحيث يحتوي كل سطر على اسم واحد، ويبدأ بالاسم الأخير، ثم الاسم الأول، أي بالصورة: `Lastname, Firstname`.

إذا أردنا التبديل بين تلك الأسماء وحذف الفاصلة الأجنبية التي بين كل منها لنحصل على الصورة `Firstname Lastname`، فنستطيع استخدام الشيفرة التالية:

```
console.log(
  "Mohsin, Samira\nFady, Eslam\nSahl, Hasan"
    .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Samira Mohsin
//   Eslam Fady
//   Hasan Sahl
```

يشير كل من $1، و$2 في السلسلة البديلة إلى المجموعات المحاطة بأقواس في النمط، ويحل النص الذي يطابق المجموعة الأولى محل $1، كما يحل النص الذي يطابق المجموعة الثانية محل $2، وهكذا حتى نصل إلى $9؛ أما التطابق كله فيمكن الإشارة إليه باستخدام $&.

من الممكن تمرير دالة بدلًا من سلسلة نصية على أساس وسيط ثاني إلى التابع replace، حيث تُستدعى الدالة لكل استبدال مع المجموعات المطابَقة والتطابق كله على أساس وسائط arguments، وتُدخَل قيمتها المعادة في السلسلة الجديدة، أي كما في المثال التالي:

```
let s = "the cia and fbi";
console.log(s.replace(/\b(fbi|cia)\b/g,
            str => str.toUpperCase()));
// → the CIA and FBI
```

وهذا مثال آخر:

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) { // only one left, remove the 's'
    unit = unit.slice(0, unit.length - 1);
  } else if (amount == 0) {
    amount = "no";
  }
  return amount + " " + unit;
}
console.log(stock.replace(/(\d+) (\w+)/g, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

يأخذ المثال أعلاه سلسلةً نصيةً، ويبحث عن حالات حدوث عدد متبوع بكلمة أبجدية رقمية، ويُعيد سلسلةً نصيةً، إذ تكون في كل حالة من تلك الحالات أُنقصت بمقدار 1.

ستكون المجموعة (\d+) هي الوسيط amount للدالة، وتقيِّد المجموعة (\w+) بالوسيط unit، وتحوِّل الدالة الوسيط amount إلى عدد، وينجح ذلك بما أنه طابَق \d+، كما تُجري بعض التعديلات في حالة إذا كان المتبقي صفر أو واحد فقط.

9.13 الجشع Greed

من الممكن استخدام `replace` لكتابة دالة تحذف جميع التعليقات من شيفرة جافاسكربت، لننظر في محاولة أولية لها:

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
console.log(stripComments("x = 10;// ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1  1
```

يطابق الجزء الذي يسبق العامل or محرفي شرطة مائلة متبوعتين بعدد من محارف لا تكون محارف سطر جديد؛ أما الجزء المتعلِّق بالتعليقات متعددة الأسطر فيملك بعض التفصيل، إذ نستخدم `[^]` -أيّ محرف ليس ضمن مجموعة المحارف الفارغة- على أساس طريقة لمطابقة أي محرف، غير أننا لا نستطيع استخدام محرف النقطة هنا لاحتواء التعليقات الكتلية على عدة أسطر، ولا يطابق محرف النقطة محارف السطر الجديد.

لكن إذا نظرنا إلى خرج السطر الأخير فسنرى أنه خطأ نوعًا ما، وذلك أنّ الجزء `*[^]` من التعبير سيطابِق كل ما يستطيع مطابقته كما وضحنا في القسم الخاص بالتعقب الخلفي، فإذا تسبب ذلك في فشل الجزء التالي من التعبير، فسيعود المطابِق محرفًا واحدًا إلى الوراء، ثم يحاول مرةً أخرى من هناك، وهو في هذا المثال يحاول مطابقة بقية السلسلة، ثم يعود إلى الوراء من هناك.

سيجد الحدث `*/` بعد العودة أربعة محارف إلى الوراء ويطابقها، وليس هذا ما أردنا، إذ كنا نريد مطابقة تعليق واحد، وليس العودة إلى نهاية الشيفرة لنجد نهاية آخر تعليق كتلي.

نقول بسبب هذا السلوك أنّ عوامل التكرار مثل `+` و`*` و`?` و`{}` هي عوامل جشعة greedy، أي تطابق كل ما تستطيع مطابقته وتتعقب خلفيًا من هناك، لكن إذا وضعنا علامة استفهام بعد تلك العوامل لتصير هكذا `+?` و `*?` و `??` و `{}?`، فسننفي عنها صفة الجشع لتطابق أقل ما يمكن، ولا تطابِق أكثر إلا كان النمط الباقي لا يناسب تطابقًا أصغر.

هذا هو عين ما نريده في هذه الحالة، فبجعل عامل النجمة يطابق أصغر امتداد من المحارف التي تقودنا إلى `*/`، فسنستهلك تعليقًا كتليًا واحدًا فقط.

```
function stripComments(code) {
return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

نستطيع على ذلك نسْب الكثير من الزلات البرمجية bugs في برامج التعابير النمطية إلى استخدام عامل جشِع من غير قصد، في حين أنّ استخدام عامل غير جشِع أفضل، لهذا يجب النظر في استخدام النسخة الغير جشعة من العامل أولًا عند استخدام أحد عوامل التكرار.

9.14 إنشاء كائنات RegExp ديناميكيا

ستكون لدينا حالات لا نعرف فيها النمط الذي يجب مطابقته عند كتابة الشيفرة، فمثلًا، نريد البحث عن اسم المستخدِم في جزء من النص، وإحاطته بمحرفي شرطة سفلية لإبرازه عما حوله، إذ لن نستطيع استخدام الترميز المبني على الشرطة المائلة لأننا لن نعرف الاسم إلا عند تشغيل البرنامج فعليًا، لكن نستطيع رغم ذلك بناء سلسلة نصية، واستخدام باني RegExp عليها، كما في المثال التالي:

```
let name = "Saad";
let text = "Saad is a suspicious character.";
let regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Saad_ is a suspicious character.
```

يجب استخدام شرطتين خلفيتين مائلتين عند إنشاء علامات الحدود \b، وذلك لعدم كتابتها في تعبير نمطي محاط بشرطات مائلة، وإنما في سلسلة نصية عادية، كذلك يحتوي الوسيط الثاني للباني RegExp على خيارات للتعبير النمطي، وهي "gi" في هذه الحالة لتشير إلى عمومها global وعدم حساسيتها لحالة المحارف case insensitive.

إذا احتوى اسم المستخدِم على محارف غريبة مثل "dea+hl[]rd"، فسيتسبب هذا في تعبير نمطي غير منطقي، وسنحصل على اسم مستخدِم لا يطابق اسم المستخدِم الفعلي.

سنضيف شرطات مائلة خلفية قبل أي محرف يملك معنىً خاص به لحل هذه المشكلة.

```
let name = "dea+hl[]rd";
let text = "This dea+hl[]rd guy is super annoying.";
let escaped = name.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("\\b" + escaped + "\\b", "gi");
```

```
console.log(text.replace(regexp, "_$&_"));
// → This _dea+hl[]rd_ guy is super annoying.
```

9.15 التابع search

لا يمكن استخدام تعبير نمطي لاستدعاء التابع indexOf على سلسلة نصية، والحل هو استخدام التابع search الذي يتوقع تعبيرًا نمطيًا، حيث يُعيد أول فهرس يجده التعبير كما يفعل indexOf، أو يُعيد -1 إذا لم يجده.

```
console.log("  word".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

لا توجد طريقة في التابع search لاختيار بدء المطابقة عند إزاحة offset بعينها، على عكس indexOf الذي يملكها في الوسيط الثاني، إذ تُعَدّ مفيدةً في بعض الأحيان.

9.16 خاصية lastIndex

لا يوفر التابع exec طريقةً سهلةً لبدء البحث من موضع بعينه في السلسلة النصية، شأنه في ذلك شأن التابع search، والطريقة التي يوفرها لذلك موجودة، لكنها ليست سهلة.

تحتوي كائنات التعبير النمطي على خصائص إحداها هي source التي تحتوي على السلسلة التي أُنشئ منها التعبير، وثمة خاصية أخرى هي lastIndex التي تتحكم في موضع بدء التطابق التالي، وإن كان في حالات محدودة، وحتى حينئذ يجب أن يكون كل من الخيار العام g وخيار y المثبت أو اللاصق sticky مفعَّلين في التعبير النمطي، كما يجب وقوع التطابق من خلال التابع exec. كان من الممكن هنا السماح بتمرير وسيط إضافي إلى exec.

```
let pattern = /y/g;
pattern.lastIndex = 3;
let match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

إذا نجح التطابق، فسيحدِّث الاستدعاء إلى exec خاصية lastIndex إلى النقطة التي تلي التطابق تلقائيًا؛ أما إذا لم يُعثر على تطابق، فستُضبِط خاصية lastIndex على الصفر، حيث يكون هو القيمة في كائن التعبير النمطي الذي سيُبنى تاليًا.

يكون الفرق بين الخيارين العام واللزج هو عدم نجاح التطابق حالة الخيار اللزج إلا عندما يبدأ من lastIndex مباشرةً؛ أما في حالة الخيار العام، فسيبحث عن موضع يمكن بدء التطابق عنده.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

تتسبب تلك التحديثات التلقائية لخاصية lastIndex في مشاكل عند استخدام قيمة تعبير نمطي مشتركة لاستدعاءات exec متعددة، فقد يبدأ تعبيرنا النمطي عند فهرس من مخلَّفات استدعاء سابق.

```
let digit = /\d/g;
console.log(digit.exec("here it is: 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

كذلك من الآثار اللافتة للخيار العام أنه يغيِّر الطريقة التي يعمل بها التابع match على السلاسل النصية، إذ يبحث عن جميع تطابقات النمط في السلسلة النصية، ويُعيد مصفوفةً تحتوي على السلاسل المطابَقة عندما يُستدعى مع الخيار العام، بدلًا من إعادة مصفوفة تشبه التي يُعيدها exec.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

لهذا يجب الحذر عند التعامل مع التعابير النمطية العامة، واستخدامها في الحالات الضرورية فقط، مثل الاستدعاءات إلى replace والأماكن التي تريد استخدام lastIndex فيها صراحةً.

9.16.1 التكرار على التطابقات

يُعَدّ البحث في جميع مرات حدوث النمط في سلسلة نصية بطريقة تعطينا وصولًا إلى كائن المطابقة في متن الحلقة loop body أمرًا شائعًا، ويمكن فعل ذلك باستخدام lastIndex و exec.

```
let input = "A string with 3 numbers in it... 42 and 88.";
let number = /\b\d+\b/g;
let match;
while (match = number.exec(input)) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
//   Found 42 at 33
//   Found 88 at 40
```

يستفيد هذا من كون قيمة تعبير الإسناد = هي القيمة المسندَة، لذا ننفذ التطابق عند بداية كل تكرار باستخدام `match = number.exec(input)` على أساس شرط في تعليمة `while`، ثم نحفظ النتيجة في رابطة binding، ونوقف التكرار إذا لم نجد تطابقات أخرى.

9.17 تحليل ملف INI

لنقل أننا نكتب برنامجًا يجمع بيانات عن أعدائنا على الإنترنت، رغم أننا لن نكتبه حقًا وإنما يهمنا الجزء الذي يقرأ ملف التهيئة، على أساس مثال على مشكلة تحتاج إلى التعابير النمطية، إذ سيبدو ملف التهيئة كما يلي:

```
searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7

; تُسبق التعليقات بفاصلة منقوطة...
; يختص كل قسم بعدو منفصل
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
fullname=Davaeorn
type=evil wizard
outputdir=/home/marijn/enemies/davaeorn
```

تكون القواعد الحاكمة لهذه الصيغة -وهي صيغة مستخدمة بكثرة، ويطلق عليها اسم INI- كما يلي:

- تُتجاهل الأسطر الفارغة والأسطر البادئة بفاصلة منقوطة.

- تبدأ الأسطر المغلَّفة بالقوسين المعقوفين [ ] قسمًا جديدًا.
- تضيف الأسطر التي تحتوي على معرِّف أبجدي-رقمي متبوع بمحرف = إعدادًا setting إلى القسم الحالي.
- لا يُسمح بأي شيء غير ما سبق، ويُعَدّ ما سواه غير صالح.

مهمتنا هنا هي تحويل سلسلة نصية مثل هذه إلى كائن تحمل خصائصه سلاسل نصية للإعدادات المكتوبة قبل ترويسة القسم الأول، وكائنات فرعية للأقسام، بحيث يحمل كل كائن فرعي إعدادات القسم الخاص به، وبما أنه يجب معالجة الصيغة سطرًا سطرًا، فمن الجيد تقسيم الملف إلى أسطر منفصلة، مستفيدين من التابع `split` الذي تعرضنا له في الفصل الرابع.

لا تقتصر بعض أنظمة التشغيل على محرف السطر الجديد لفصل الأسطر، وإنما تستخدِم محرف العودة لبداية السطر carriage return متبوعًا بسطر جديد `"\r\n"`، وبما أنّ التابع `split` يسمح بالتعبير النمطي على أساس وسيط، فنستطيع استخدام تعبير نمطي مثل `/\r?\n/` للتقسيم بطريقة تسمح بوجود كل من `"\n"` و `"\r\n"` بين الأسطر.

```
function parseINI(string) {
  // ابدأ بكائن ليحمل حقول المستوى العلوي
  let result = {};
  let section = result;
  string.split(/\r?\n/).forEach(line => {
    let match;
    if (match = line.match(/^(\w+)=(.*)$/)) {
      section[match[1]] = match[2];
    } else if (match = line.match(/^\[(.*)\]$/)) {
      section = result[match[1]] = {};
    } else if (!/^\s*(;.*)?$/.test(line)) {
      throw new Error("Line '" + line + "' is not valid.");
    }
  });
  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
```

```
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

تمر الشيفرة على أسطر الملف وتبني الكائن، كما تخزِّن الخصائص التي في القمة داخل الكائن مباشرةً، بينما تخزن الخصائص الموجودة في الأقسام داخل كائن قسم مستقل، كما تشير الرابطة `section` إلى كائن القسم الحالي.

لدينا نوعان من الأسطر المميزة، وهما ترويسة الأقسام، أو أسطر الخصائص، فإذا كان السطر خاصيةً عاديةً، فسيخزَّن في الموضع الحالي؛ أما إذا كان ترويسةً لقسم، فسيُنشأ كائن قسم جديد وتُضبط `section` لتشير إليه.

نضمن بالاستخدام المتكرر لمحرفي `^` و`$`، مطابقة التعبير للسطر كاملًا وليس جزءًا منه فقط، كما ستعمل الشيفرة عند إهمالهما، لكنها ستتصرف مع بعض المدخلات بغرابة، وهو الأمر الذي سيكون زلةً bug يصعب تعقبها وإصلاحها.

يُعَدّ النمط `if (match = string.match(...))` شبيهًا بما سبق في شأن استخدام الإسناد على أساس شرط لتعليمة `while`، فلن نكون على يقين من نجاح استدعاء `match`، لذا لا نستطيع الوصول إلى الكائن الناتج إلا داخل تعليمة `if` تختبر ذلك، ولكي لا نقطع سلسلة الصيغ `else if`، فسنسند نتيجة التطابق إلى رابطة، وسنستخدم هذا التعيين على أساس اختبار لتعليمة `if` مباشرةً.

تتحقق الدالة من السطر باستخدام التعبير `/^\s*(;.*)?$/` إذا لم يكن ترويسةً لقسم أو خاصيةً ما، إذ تتحقق من أنه تعليق أو سطر فارغ، حيث يطابق الجزء الذي بين الأقواس التعليقات، ثم تتأكد `?` أنه يطابق الأسطر التي تحتوي على مسافة بيضاء فقط، وإذا وُجد سطر لا يطابق أي صيغة من الصيغ المتوقعة، فسترفع الدالة استثناءً exception.

9.18 المحارف الدولية

كان اتجاه تصميم جافاسكربت في البداية نحو سهولة الاستخدام، وقد ترسخ ذلك الاتجاه مع الوقت إلى أن صار هو السمة الأساسية للغة ومعيارًا لتحديثاتها، لكن أتت هذه السهولة بعواقب لم تكن في الحسبان وقتها، إذ تُعَدّ تعابير جافاسكربت النمطية غبيةً لغويًا، فالمحرف الكلمي بالنسبة لها هو واحد من 26 محرفًا فقط، وهي المحارف الموجودة في الأبجدية اللاتينية بحالتيها الصغرى والكبرى، أو أرقامًا عشريةً، أو محرف الشرطة السفلية _ أما بالنسبة لأيّ شيء غير ذلك، مثل é أو β، فلن تطابق `\w` رغم أنها محارف كلمية، لكنها ستطابق الحالة الكبرى منها `\W` التي تشير إلى التصنيف غير الكلمي nonword category.

تجدر الإشارة إلى ملاحظة غريبة في شأن محرف المسافة البيضاء العادية `\s`، إذ لا تعاني من هذه المشكلة، وتطابق جميع المحارف التي يَعُدّها معيار اليونيكود محارف مسافة بيضاء، بما في ذلك المسافة غير الفاصلة nonbreaking space، والفاصل الصوتي في اللغة المنغولية Mongolian vowel separator.

أيضًا، من المشاكل التي سنواجهها مع التعابير النمطية في جافاسكربت أنها لا تعمل على المحارف الحقيقية وإنما تعمل على الأعداد البِتّية للمحارف code units كما ذكرنا في الفصل الخامس، وبالتالي سيكون سلوك المحارف المكونة من عددين بِتّيين غريبًا، وعلى خلاف ما نريد.

```
console.log(/🍏{3}/.test("🍏🍏🍏"));
// → false
console.log(/<.>/.test("<🌹>"));
// → false
console.log(/<.>/u.test("<🌹>"));
// → true
```

المشكلة أنّ 🍏 التي في السطر الأول تعامَل على أنها عددين بِتّيين، ولا يطبَّق الجزء `{3}` إلا على العدد الثاني. وبالمثل، تطابق النقطة عددًا بِتَيًا واحدًا، وليس العددين اللذَين يكونان الرمز التعبيري للوردة، كما يجب إضافة خيار اليونيكود `u` للتعبير النمطي كي يعامَل مثل تلك المحارف على الوجه الذي ينبغي.

سيظل السلوك الخاطئ للتعبير النمطي هو الافتراضي للأسف، لأنّ التغيير قد يتسبب في مشاكل للشيفرة الموجودة والتي تعتمد عليه، ومن الممكن استخدام `\p` في تعبير نمطي مفعّل فيه خيار اليونيكود لمطابقة جميع المحارف التي يسند اليونيكود إليها خاصية معطاة، رغم أنّ هذه الطريقة معتمدة حديثًا ولم تُستخدم كثيرًا بعد.

```
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

يعرِّف اليونيكود عددًا من الخصائص المفيدة رغم أنّ إيجاد الخاصية التي نحتاجها قد لا يكون أمرًا سهلًا في كل مرة، حيث يمكن استخدام الصيغة `\p{Property=Value}` لمطابقة أيّ محرف له قيمة معطاة لتلك الخاصية، وإذا أُهمل اسم الخاصية كما في `\p{Name}`، فسيُفترض الاسم إما خاصيةً بِتّيةً مثل `Alphabetic`، أو فئةً مثل `Number`.

9.19 خاتمة

التعبيرات النمطية هي كائنات تمثل أنماطًا في السلاسل النصية، وتستخدم لغتها الخاصة للتعبير عن تلك الأنماط.

التعبير النمطي	دلالته
`/abc/`	تسلسل من المحارف
`/[abc]/`	أيّ محرف في مجموعة محارف
`/[^abc]/`	أيّ محرف ليس في مجموعة ما من المحارف
`/[0-9]/`	أيّ محرف من مجال ما من المحارف
`/x+/`	مرة حدوث واحدة أو أكثر للنمط `x`
`/x+?/`	مرة حدوث أو أكثر غير جشعة
`/x*/`	حدوث صفري أو أكثر.
`/x?/`	حدوث صفري أو حدوث لمرة واحدة
`/x{2,4}/`	حدوث لمرتَين إلى أربعة مرات
`/(abc)/`	مجموعة
`/a|b|c/`	أيّ نمط من بين أنماط متعددة
`/\d/`	أيّ محرف رقمي
`/\w/`	محرف أبجدي رقمي alphanumeric، أي محرف كلمة
`/\s/`	أيّ محرف مسافة بيضاء
`/./`	أيّ محرف عدا الأسطر الجديدة
`/\b/`	حد كلِمي word boundary
`/^/`	بداية الدخل
`/$/`	نهاية الدخل

يملك التعبير النمطي التابع `test` للتحقق هل السلسلة المعطاة مطابقة أم لا، كما يملك التابع `exec` الذي يُعيد مصفوفةً تحتوي على جميع المجموعات المطابِقة إذا وُجدت مطابقات، ويكون لتلك المصفوفة خاصية `index` التي توضِّح أين بدأت المطابقة.

تملك السلاسل النصية التابع `match` الذي يطابقها مع تعبير نمطي، وتابع `search` الذي يبحث عن التعابير النمطية ثم يُعيد موضع بداية التطابق فقط، كما تملك السلاسل النصية تابعًا اسمه `replace`، حيث يستبدِل سلسلةً نصيةً أو دالةً بتطابقات النمط.

يمكن امتلاك التعابير النمطية خيارات تُكتب بعد شرطة الإغلاق المائلة، إذ يجعل الخيار `i` التطابق حساسًا لحالة الأحرف، كما يجعل الخيار `g` التعبير عامًا global، ويمكّن التابع `replace` من استبدال جميع النسخ بدلًا من النسخة الأولى فقط؛ أما الخيار `y` فيجعله لزجًا، أي لن يتجاوز جزءًا من السلسلة أثناء البحث عن تطابق، كذلك يفعّل الخيار `u` وضع اليونيكود الذي يصلح لنا عددًا من المشاكل المتعلقة بمعالجة المحارف التي تأخذ أكثر من عددين بتيين.

وهكذا فإنّ التعابير النمطية أشبه بسكين حاد لها مقبض غريب الشكل، فهي تيسِّر المهام التي ننفذها كثيرًا، لكن قد تصبح صعبة الإدارة حين نستخدمها في مشاكل معقدة، ومن الحكمة تجنب حشر الأشياء التي لا تستطيع التعبيرات النمطية التعبير عنها بسهولة.

9.20 تدريبات

ستجد نفسك لا محالةً أثناء العمل على هذه التدريبات أمام سلوكيات محيِّرة للتعابير النمطية، فمن المفيد عندئذ إدخال تعبيراتك النمطية في أداة مثل https://debuggex.com لترى إن كان تصورها المرئي يوافق السلوك الذي أردت أم لا، ولترى كيف تستجيب لسلاسل الدخل المختلفة.

9.20.1 Regexp golf

Code golf هو مصطلح مستخدم للعبة تحاول التعبير عن برنامج معيَّن بأقل عدد ممكن من المحارف، وبالمثل، يكون regexp golf عملية كتابة تعابير نمطية، بحيث تكون أصغر ما يمكن، وتطابق النمط المعطى فقط. اكتب تعبيرًا نمطيًا لكل عنصر مما يلي، بحيث يتحقق من حدوث أيّ سلسلة نصية فرعية داخل السلسلة النصية الأم، ويجب على التعبير النمطي مطابقة السلاسل المحتوية على إحدى السلاسل الفرعية التي ذكرناها. لا تشغل بالك بحدود الكلمات إلا إذا ذُكر ذلك صراحةً، وإذا نجح تعبيرك النمطي فانظر إن كنت تستطيع جعله أصغر.

1. Car وcat.
2. Pop وprop.
3. Ferret وferry وferrari.
4. أيّ كلمة تنتهي بـ ious.
5. محرف مسافة بيضاء متبوع بنقطة، أو فاصلة أجنبية، أو نقطتين رأسيتين، أو فاصلة منقوطة.
6. كلمة أكبر من ستة أحرف.
7. كلمة ليس فيها الحرف e أو E.

استرشد بالجدول الذي في خاتمة الفصل أعلاه، واختبر كل حل ببعض السلاسل النصية.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفيـة المتصـفح إن كنت تقـرأ من متصـفح، أو بنسخها إلى codepen.

```
// املأ التعابير النمطية التالية

verify(/.../,
       ["my car", "bad cats"],
       ["camper", "high art"]);

verify(/.../,
       ["pop culture", "mad props"],
       ["plop", "prrrop"]);

verify(/.../,
       ["ferret", "ferry", "ferrari"],
       ["ferrum", "transfer A"]);

verify(/.../,
       ["how delicious", "spacious room"],
       ["ruinous", "consciousness"]);

verify(/.../,
       ["bad punctuation ."],
       ["escape the period"]);

verify(/.../,
       ["Siebentausenddreihundertzweiundzwanzig"],
       ["no", "three small words"]);

verify(/.../,
       ["red platypus", "wobbling nest"],
       ["earth bed", "learning ape", "BEET"]);


function verify(regexp, yes, no) {
  // تجاهل التدريبات غير المكتملة
  if (regexp.source == "...") return;
```

```
  for (let str of yes) if (!regexp.test(str)) {
    console.log(`Failure to match '${str}'`);
  }
  for (let str of no) if (regexp.test(str)) {
    console.log(`Unexpected match for '${str}'`);
  }
}
```

9.20.2 أسلوب الاقتباس

تخيَّل أنك كتبت قصةً، واستخدمت علامات الاقتباس المفردة فيها لتحديد النصوص التي قالتها الشخصيات فيها، وتريد الآن استبدال علامات الاقتباس المزدوجة بكل تلك العلامات المفردة، لكن مع استثناء الكلمات التي تكون فيها العلامة المفردة لغرض مختلف مثل كلمة aren't.

فكر في نمط يميز هذين النوعين من استخدامات الاقتباس، وصمم استدعاءً إلى التابع `replace` الذي ينفذ عملية الاستبدال المناسبة. تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
let text = "'أنا الطاهي،' he said, 'إنها وظيفتي.' ";
// غيِّر هذا الاستدعاء.
console.log(text.replace(/A/g, "B"));
// → "أنا الطاهي،" he said, "إنها وظيفتي."
```

إرشادات الحل

يكون الحل البديهي هنا هو استبدال محرف غير كلمي nonword بعلامات الاقتباس على الأقل من جانب واحد، كما في `/\W'|'\W/`، لكن سيكون عليك أخذ بداية السطر ونهايته في حسابك.

كذلك يجب ضمان أنّ الاستبدال سيشمل المحارف التي طابقها نمط `\W` كي لا تُنسى، ويمكن فعل ذلك بتغليفها في أقواس، ثم تضمين مجموعاتها في السلسلة النصية البديلة (`$1` و`$2`)، كما لا يُستبدل شيء بالمجموعات التي لم تطابَق.

9.20.3 الأعداد مرة أخرى

اكتب تعبيرًا لا يطابق إلا الأعداد التي لها نسق جافاسكربت، ويجب عليه دعم علامة + أو - قبل العدد، والفاصلة العشرية، والصيغة الأسية -أي `5e-3`، أو `1E10`-، مع علامتي موجب أو سالب قبل الأس.

لاحظ عدم اشتراط وجود أرقام قبل فاصلة العشرية أو بعدها، لكن لا يمكن أن يكون العدد مكونًا من فاصلة العشرية وحدها، أي يسمح بكل من `5.` و`.5` في جافاسكربت، لكن لا يُسمح بـ `..`

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// املأ التعبير النمطي التالي.
let number = /^...$/;

// الاختبارات:
for (let str of ["1", "-1", "+15", "1.55", ".5", "5.",
                 "1.3e2", "1E-4", "1e+12"]) {
  if (!number.test(str)) {
    console.log(`Failed to match '${str}'`);
  }
}
for (let str of ["1a", "+-1", "1.2.3", "1+1", "1e4.5",
                 ".5.", "1f5", "."]) {
  if (number.test(str)) {
    console.log(`Incorrectly accepted '${str}'`);
  }
}
```

إرشادات الحل

- يجب عدم نسيان الشرطة المائلة الخلفية التي قبل النقطة.
- يمكن مطابقة العلامة الاختيارية التي قبل العدد وقبل الأس بواسطة `[+\-]?`، أو `(\+|-|)`، والتي تعني موجب، أو سالب، أو لا شيء.
- يبقى الجزء الأصعب من هذا التدريب مطابقة كل من `"5."`، و `".5"` دون مطابقة `"."`، وأحد الحلول الممتازة هنا هو استخدام العامل `|` لفصل الحالتين؛ فإما رقم واحد أو أكثر متبوع اختياريًا بنقطة وصفر، أو أرقام أخرى، أو نقطة متبوعة برقم واحد أو أكثر.
- وأخيرًا، نريد جعل e حساسة لحالتها؛ فإما نضيف الخيار `i` للتعبير النمطي أو نستخدم `[eE]`.

دورة تطوير التطبيقات
باستخدام لغة JavaScript

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

10. الوحدات Modules

لا تكتب شيفرةً يسهل توسيعها فيما بعد، بل اكتبها بحيث يسهل محوها.

— توماس فِج Thomas Figg، كتاب "البرمجة سيئة" Programming is Terrible.

إذا أردنا وصف البرنامج المثالي، فسيكون ذلك الذي له بنية واضحة مثل الكريستال، ويسهل وصف طريقة عمله، كما يملك كل جزء فيه دورًا محدَّدًا ومعروفًا.

لكن هذا لا يكون في الواقع، بل تكون لدينا برامج تنمو نموًا عضويًا، بحيث تضاف إليها المزايا كلما دعت الحاجة إلى ذلك؛ أما الهيكلة البنائية لها والحفاظ على تلك الهيكلية فهي أمر آخر، إذ لا تُرى ثمرتها إلا في المستقبل حين يأتي شخص آخر ليعمل على البرنامج، وعلى ذلك فمن المغري للمبرمج إهمالها الآن لتصبح أجزاء البرنامج متشعبةً ومتداخلةً إلى حد فظيع.

يتسبب هذا في مشكلتين حقيقيتَين، أولاهما أنّ فهم مثل هذا النظام صعب جدًا، إذ سيكون من العسير النظر لأيّ جزء فيه نظرة مستقِلة إذا كان كل شيء فيه يتصل بشيء آخر، حيث سنُجبَر على دراسة البرنامج كله من أجل فهم جزء واحد فقط؛ أما الثانية فهي إذا أردنا استخدام أيّ وظيفة من مثل هذا البرنامج في حالة أخرى، فستكون إعادة كتابتها من الصفر أسهل من استخراجها مستقِلةً من شيفرتها لتعمل في مكان آخر.

يستخدم مصطلح "كرة الوحل" لمثل تلك البرامج التي لا هيكل لها، والتي يلتصق كل شيء فيها ببعضه، فإذا حاولت استخراج جزء، فستنهار الكرة كلها، ولا ينالك إلا تلطيخ يدك.

10.1 الوحدات Modules

تُعَدّ الوحدات محاولةً لتفادي مثل تلك المشاكل التي وصفناها، فالوحدة هي جزء من برنامج ما يحدِّد بوضوح ما هي الأجزاء الأخرى التي يعتمد عليها، وما هي الوظيفة -أي الواجهة الخاصة به- التي سيوفرها للوحدات الأخرى لتستخدمها.

تشترك واجهات الوحدات مع واجهات الكائنات في أمور كثيرة كما رأينا في الفصل السادس، إذ تجعل جزءًا من الوحدة متوفرًا للعالم الخارجي وتحافظ على خصوصية الباقي، كما يصبح النظام أشبه بقِطع الليجو LEGO في تقييد الطرق التي تتفاعل الوحدات بها مع بعضها البعض، فتتفاعل الأجزاء المختلفة من خلال وصلات connectors معرَّفة جيدًا، على عكس الوحل الذي يختلط فيه كل شيء.

تسمى العلاقات التي بين الوحدات بالاعتماديات dependencies، فإذا احتاجت وحدة إلى جزء من وحدة أخرى، فسيقال أنها تعتمد على تلك الوحدة، وحين توصف تلك الحقيقة بوضوح داخل الوحدة نفسها، فيمكن استخدامها لمعرفة الوحدات الأخرى التي يجب أن تكون موجودةً من أجل استخدام وحدة ما، ولتحميل الاعتماديات تلقائيًا، كما ستحتاج كل واحدة منها إلى مجال خاص private scope إذا أردنا فصل الوحدات بهذه الطريقة.

لا يكفي وضع شيفرة جافاسكربت في ملفات مختلفة لتلبية تلك المتطلبات، فلا زالت الملفات تتشارك فضاء الاسم العام global namespace نفسه، كما قد تتداخل عن قصد أو غير قصد مع رابطات بعضها bindings، وهكذا تظل هيكلية الاعتماديات غير واضحة، وسنرى بعد قليل كيف نعالج ذلك.

قد يكون التفكير في تصميم وحدة ذات هيكل مناسب لبرنامج ما صعبًا بما أننا في مرحلة البحث عن المشكلة واستكشاف حلول لها، وذلك لنرى أيها يعمل، كما لا نريد شغل بالنا بتنظيم أحد الحلول إلا حين يثبت نجاحه.

10.2 الحزم packages

نستطيع استخدام نفس الجزء في برامج أخرى مختلفة، وهي إحدى المزايا التي نحصل عليها عند بناء برنامج من أجزاء منفصلة تستطيع العمل مستقِلة عن بعضها البعض.

لكن كيف نُعِدّ هذا؟ لنقل أننا نريد استخدام الدالة `parseINI` من الفصل التاسع في برنامج آخر، فإذا كان ما تعتمد عليه الدالة واضحًا -لا تعتمد الدالة على شيء في هذه الحالة-، فلن نفعل أكثر من نسخ شيفرتها إلى المشروع الجديد واستخدامها؛ أما إذا رأينا خطأً في تلك الشيفرة، فسنصلحه في ذلك المشروع الجديد وننسى إصلاحه في البرنامج الأول الذي أخذنا منه الشيفرة، وبهذا نهدر الوقت والجهد في نسخ الشيفرات من هنا إلى هناك والحفاظ عليها محدَّثة.

وهنا يأتي دور الحِزم packages، فهي قطعة من شيفرة يمكن نشرها وتوزيعها -أي نسخها وتثبيتها-، وقد تحتوي على وحدة واحدة أو أكثر، كما فيها معلومات عن الحِزم الأخرى التي تعتمد عليها، وتأتي مع توثيق يشرح وظيفتها كي يستطيع الناس استخدامها، فلا تكون مقتصرةً على من كتبها فقط.

تُحدَّث حزمة ما إذا وُجدت مشكلة فيها أو أضيفت إليها ميزة جديدة، وعليه تكون البرامج التي تعتمد عليها -والتي قد تكون حِزمًا أيضًا- يمكنها الترقية إلى تلك النسخة الجديدة.

يتطلب العمل بهذه الطريقة بنيةً تحتيةً، وذلك لحاجتنا إلى مكان لتخزين تلك الحِزم والبحث فيها، وإلى طريقة سهلة لتثبيتها وترقيتها كذلك، كما تُوفَّر هذه البنية التحتية في عالم جافاسكربت من قِبَل NPM وهي اختصار للعبارة Node Package Manager، التي تعني مدير الحِزم.

يتكون مدير الحِزم NPM من شيئين، أولهما تقدِّم خدمة تنزيل ورفع الحزم عبر الإنترنت -أي أونلاين online-، وبرنامج -مضمَّن مع Node.js- يساعدك على تثبيت تلك الحزم وإدارتها، كما توجد أكثر من نصف مليون حزمة متاحة على NPM وقت كتابة هذه الكلمات، وأكثرها لا فائدة منه، لكن الحزم المفيدة المتاحة للعامة موجودة هناك أيضًا.

سنجد محلل ملفات INI مثلًا الذي يشبه ما بنيناه في الفصل التاسع في هيئة حزمة اسمها `ini`، كما سنرى في الفصل العشرين كيف نُثبِّت مثل تلك الحزم محليًا باستخدام أمر `npm` في الطرفية.

إتاحية مثل تلك الحزم عالية الجودة للتحميل مفيد جدًا، فهو يعني استطاعتنا تجنب إعادة اختراع برنامج كتبه مئة إنسان قبلنا، كما نتجاوز ذلك إلى استخدام حزمة مجرَّبة ومختبَرة جيدًا ببضع ضغطات على لوحة المفاتيح، ومن بداهة القول أنّ البرامج لا تكلف شيئًا في نَسخها، لكن كتابة البرامج أول مرة هي العمل الذي يكلف الجهد والوقت والمهارة، كما يماثلها الاستجابة لمن يجد مشاكل في الشيفرة، أو الاستجابة لمن يريد إدخال ميزات جديدة في البرنامج.

مَن يكتب البرنامج يمتلك حقوقه افتراضيًا، ولا يستطيع أحد استخدامه إلا برخصة من المبرمج، لكن بما أنّ بعض البشر ذوو قلوب طيبة، ولأنّ نشر البرامج الجيدة سيبني لك سُمعة وسط المبرمجين، فستسمح صراحةً كثير من الحِزم تحت رخصة ما باستخدامها من أيّ كان.

ترخَّص أغلب الشيفرات الموجودة في NPM بتلك الطريقة، في حين قد تشترط بعض الرخص نشر الشيفرة التي تبنيها على قمة الحزمة التي استخدمتها تحت رخصة الحزمة نفسها، وأخرى لا تريد أكثر من استخدام رخصة الحزمة نفسها حين تنشر برنامجك، وهي الرخصة التي يستخدمها أغلب مجتمع جافاسكربت، وبالتالي تأكد حين تستخدِم حِزمًا كتبها غيرك من قراءة الرخصة التي تأتي بها الحزمة.

10.3 الوحدات المرتجلة Improvised modules

لم يكن في جافاسكربت نظام وحدات مضمَّن حتى عام 2015، إذ لم يمنع ذلك الناس من بناء نظمًا كبيرة في جافاسكربت طيلة أكثر من عشر سنين، كما كانوا في أمسّ الحاجة إلى وحدات لهذا، وعليه فقد صمموا نظم وحداتهم الخاصة بهم فوق اللغة نفسها، حيث نستطيع استخدام دوال جافاسكربت لإنشاء نطاقات محلية local scopes وكائنات لتمثل واجهات الوحدات.

لدينا فيما يلي وحدةً للانتقال بين أسماء الأيام وأرقامها -وذلك من إعادة التابع `getDay` الخاص بـ `Date`-، إذ تتكون واجهتها من `weekDay.name` و`weekDay.number`، كما تخفي رابطتها `names` المحلية داخل نطاق تعبير الدالة الذي يُستدعى فورًا.

```
const weekDay = function() {
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

يوفِّر هذا النسق من الوحدات عزلًا إلى حد ما، لكنه لا يصرح عن الاعتماديات، كما يضع واجهته في النطاق العام global scope، ويَتوقع اعتمادياتها إذا وُجدت أن تحذو حذوه، في حين كان ذلك هو الأسلوب المتَّبع في برمجة الويب لمدة طويلة، لكنه صار مهجورًا الآن وقلّما يُستخدَم.

إذا أردنا جعل علاقات الاعتماديات جزءًا من الشيفرة، فيجب التحكم في تحميل الاعتماديات، ويتطلب ذلك أن نستطيع تنفيذ السلاسل النصية على أساس شيفرات، إذ تستطيع جافاسكربت فعل ذلك لحسن الحظ.

10.4 تقييم البيانات على أساس شيفرات

هناك عدة طرق لأخذ البيانات -أي سلسلة نصية من شيفرات برمجية- وتشغيلها على أساس جزء من البرنامج الحالي، وأبسط طريقة هي العامل الخاص `eval` الذي سينفِّذ السلسلة النصية في النطاق الحالي current scope، لكن هذه فكرة سيئة لا يُنصح بها، إذ تعطِّل بعض الخصائص التي تكون عادةً للمجالات مثل توقعها للرابطة التي يشير إليها اسم ما بسهولة.

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1
```

يمكن تفسير البيانات على أساس شيفرة بطريقة أبسط باستخدام الباني `Function` الذي يأخذ وسيطين هما سلسلة نصية تحتوي قائمة من أسماء الوسائط مفصول بينها بفاصلة أجنبية، وسلسلة نصية تحتوي على الدالة نفسها.

يغلف الباني الشيفرة داخل قيمة دالة كي تحصل على نطاقها الخاص، ولا تفعل أمورًا غريبة مع النطاقات الأخرى.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

هذا هو المطلوب تحديدًا وما نحتاج إليه في نظام الوحدات، إذ نستطيع تغليف شيفرة الوحدة في دالة، ونستخدم نطاق تلك الدالة على أساس نطاق الوحدة.

10.5 CommonJS

لعل أكثر منظور لوحدات جافاسكربت المضافة إليها هو نظام CommonJS (اختصارًا إلى common JavaScript أي وحدات جافاسكربت المشتركة)، إذ تستخدِمه Node.js الذي هو النظام المستخدَم في أغلب الحِزم على NPM.

تدور الفكرة العامة لوحدات CommonJS حول دالة اسمها `require`، فحين نستدعيها مع اسم وحدة الاعتمادية، فستضمن أن الوحدة محمَّلة وستعيد واجهتها.

تحصل الوحدات على نطاقها المحلي الخاص بها تلقائيًا لأنّ المحمّل يغلف شيفرة الوحدة في دالة، وما عليها إلا استدعاء `require` كي تصل إلى اعتمادياتها، وتضع واجهاتها في الكائن المقيَّد بتعليمة `exports`.

توفِّر الوحدة المثال التالية دالـةً لتنسـيق التـاريخ، إذ تسـتخدِم حزمـتين من NPM هما `ordinal` لتحويـل الأعداد إلى سلاسل نصية مثل `"1st"` و`"2nd"`، و`date-names` للحصول على الأسماء الإنجليزية للشهور وأيام الأسبوع، ثم تصدِّر دالةً وحيدةً هي `formatDate` تأخذ كائن `Date` وسلسلة قالب template string.

قـد تحتـوي سلسـلة القـالب على شـيفرات توجـه التنسـيق مثـل `YYYY` للسـنة كاملـة و`Do` لـترتيب اليـوم في الشـهر، ومن الممكن إعطاؤهـا سلسـلةً نصـيةً مثـل `"YYYY Do MMMM"` للحصـول على خـرج مثل "November 22nd 2017".

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

تتكـون واجهـة `ordinal` من دالـة واحـدة، بينمـا تصـدِّر `date-names` كائنًـا يحتـوي على عـدة أشـياء، إذ تُعَدّ `days` و`months` مصفوفات من الأسماء، كما تُعَدّ عملية فك البنية الهيكلية سهلةً جدًا عنـد إنشـاء رابطـات للواجهات المستوردة.

تضيف الوحدة كذلك دالة واجهتها إلى `exports` كي تصل إليها الوحدات الـتي تعتمـد عليهـا، إذ نسـتطيع استخدام الوحدة كما يلي:

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

نستطيع تعريف `require` في أبسط صورها كما يلي:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = {exports: {}};
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

تَقرأ الدالة المختلَقة `readFile` في المثال أعلاه ملفًا وتعيد محتوياته في سلسلة نصية؛ أما جافاسكربت فلا توفِّر مثل تلك الخاصية، لكن توفِّر بيئات جافاسكربت المختلفة مثل المتصفحات وNode.js طرقها الخاصة للوصول إلى الملفات، وقد كان المثال يفترض وجود دالة اسمها `readFile`.

تحتفظ دالة `require` بذاكرة مؤقتة cache من الوحدات المحمَّلة بالفعل، وذلك لتجنب تحميل الوحدة نفسها عدة مرات، فإذا استُدعيت، فستنظر أولًا إن كانت الوحدة المطلوبة محمَّلة من قبل أم لا، ثم تُحمِّلها إن لم تكن محمَّلة، حيث يتطلب ذلك قراءة شيفرة الوحدة وتغليفها في دالة واستدعاءها.

لم تكن واجهة حِزمة `ordinal` التي رأيناها من قبل كائنًا وإنما دالةً، ومن مزايا CommonJS أنه رغم إنشاء نظام الوحدات لكائن واجهة فارغ مقيَّد بـ `exports`، يمكننا استبدال أيّ قيمة بذلك الكائن عبر إعادة كتابة `module.exports`، حيث يتم ذلك بعدة وحدات لتصدير قيمة واحدة بدلًا من كائن واجهة؛ وإذا عرَّفنا `require` و`exports` و`module` على أساس معامِلات لدالة التغليف المولَّدة وتمرير القيم المناسبة عند استدعائها، فسيضمن المحمِّل إتاحية تلك الرابطات في نطاق الوحدة.

تختلف الطريقة التي تُرجِمت بها السلسلة النصية التي أُعطيت إلى دالة `require` إلى اسم ملف حقيقي أو عنوان ويب باختلاف الأنظمة، فإذا بدأت بـ `"./"` أو `"../"`، فستُفسَّر تفسيرًا مرتبطًا باسم ملف الوحدة، وبالتالي سيكون `"./format-date"` هو الملف المسمى `format-date.js` في المجلد نفسه؛ أما إذا لم يكن الاسم مرتبطًا بالوحدة، فستبحث Node.js عن حِزمة مثبَّتة تحمل الاسم نفسه، وسنفسر مثل تلك الأسماء التي في شيفرات أمثلة هذا الفصل على أنها تشير إلى حِزم NPM، كما سننظر بالتفصيل في كيفية تثبيت واستخدام وحدات NPM في الفصل العشرين.

نستطيع استخدام واحد من NPM الآن بدلًا من كتابة محلل ملف INI الخاص بنا.

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

10.6 وحدات ECMAScript

تظل وحدات CommonJS حلًا غير دائم وغير احترافي رغم عملها بكفاءة وسماحها لمجتمع جافاسكربت مع NPM بتشارك الشيفرات على نطاق واسع، وتُعَدّ صيغتها غريبةً قليلًا، إذ لا تكون الأشياء التي نضيفها إلى `exports` متاحةً في النطاق المحلي مثلًا، وبما أنّ `require` هي استدعاء دالة عادية تأخذ أيّ نوع من الوسطاء وليس القيم مصنَّفة النوع فقط، فمن الصعب تحديد اعتماديات الوحدة دون تشغيل شيفرتها.

لهذا يُصدِر معيار جافاسكربت نظام وحداته الخاص منذ 2015، كما يطلق عليه غالبًا ES modules، حيث تشير ES إلى ECMAScript، ورغم أنّ المفاهيم الأساسية للاعتماديات والواجهات لا زالت كما هي، إلا أنه يختلف في التفاصيل، فصارت الصيغة الآن مدمجةً في اللغة، كما نستطيع استخدام الكلمة المفتاحية الخاصة `import` بدلًا من استدعاء دالة للوصول إلى اعتمادية ما.

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

تُستخدم بالمثل كلمة `export` المفتاحية لتصدير الأشياء، وقد تأتي قبل تعريف دالة أو صنف أو رابطة (`let` أو `const` أو `var`).

لا تكون واجهة وحدة ES قيمةً واحدةً، بل مجموعة من الرابطات المسماة، كما تربط الوحدة السابقة `formatDate` بدالة، فإذا استوردنا من وحدة أخرى، فسنستورد الرابطة وليس القيمة، مما يعني أن الوحدة المصدِّرة قد تغير قيمة الرابطة في أي وقت، وسترى الوحدات المستوردة القيمة الجديدة.

إذا وُجدت رابطة اسمها `default` فستُعامَل على أنها القيمة المصدَّرة الأساسية للوحدة، فإذا استوردنا وحدةً مثل `ordinal` التي في المثال دون الأقواس التي حول اسم الرابطة، فسنحصل على رابطة `default`، كما تستطيع مثل تلك الوحدات تصدير رابطات أخرى تحت أسماء مختلفة مع تصدير `default` الخاص بها.

إذا أردنا إنشاء تصدير افتراضي، فسنكتب `export default` قبل التعبير أو تصريح الدالة أو تصريح الصنف.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

من الممكن إعادة تسمية الرابطات المستورَدة باستخدام الكلمة `as`.

```
import {days as dayNames} from "date-names";
console.log(dayNames.length);
// → 7
```

من الفروقات المهمة كذلك هو حدوث تصديرات وحدات ES قبل بدء تشغيل سكربت الوحدة، وبالتالي قد لا تظهر تصريحات `import` داخل الدوال أو الكتل، ويجب أن تكون أسماء الاعتماديات سلاسل نصية مقتَبسة وليست تعبيرات عشوائية.

يعكف مجتمع جافاسكربت أثناء كتابة هذه الكلمات على اعتماد وتبني أسلوب الوحدات هذا، لكن يبدو أنها عملية طويلة، فقد استغرقنا بضع سنين بعد الاستقرار على الصياغة كي تدعمها المتصفحات وNode.js، ورغم أنها صارت مدعومةً إلى حد كبير إلا أنّ ذلك الدعم به مشاكل، ولا زال الجدل قائمًا حول الكيفية التي يجب توزيع تلك الوحدات بها في NPM.

تُكتب مشاريع عديدة باستخدام وحدات ES، ثم تُحوَّل تلقائيًا إلى صيغة أخرى عند نشرها، فنحن في مرحلة انتقالية يُستخدم فيها نظامَي وحدات جنبًا إلى جنب، ومن المفيد أننا نستطيع قراءة وكتابة شيفرات بكليهما.

10.7 البناء والتجميع

إذا نظرنا إلى مشاريع جافاسكربت فسنجد العديد منها لا يُكتب بجافاسكربت من الأساس -تقنيًا- فثَمة امتدادات مستَخدَمة على نطاق واسع مثل ذلك الذي تعرضنا له في الفصل الثامن الذي يتحقق من اللهجة، وقد بدأت الناس باستخدام الامتدادات الجاهزة في اللغة قبل إضافتها إلى المنصات التي تشغِّل جافاسكربت بزمن طويل أصلًا، ولكي يستطيع المبرمج فعل ذلك فإنه يصرِّف compile شيفرته، ثم يترجمها من لهجة جافاسكربت التي استخدمها إلى جافاسكربت عادية، أو إلى نسخة أقدم من جافاسكربت كي تستطيع المتصفحات الأقدم تشغيلها.

لكن إدخال برنامج يتكون من وحدات ومن 200 ملف مختلف إلى صفحة ويب له مشاكله، فإذا كان جلب الملف عبر الشبكة يستغرق 50 ميلي ثانية، فسيستغرق تحميل البرنامج كله 10 ثواني، أو نصف تلك المدة إن كان يحمِّل عدة ملفات في الوقت نفسه، وهذا وقت ضائع.

بدأ مبرمجو الويب باستخدام أدوات تجمع برامجهم التي قضوا الساعات المضنية في تقسيمها إلى وحدات، ليكون البرنامج ملفًا واحدًا كبيرًا، ثم ينشرونه على الويب، ذلك أن جلب ملف واحد وإن كان كبيرًا عبر الويب سيكون أسرع من جلب الكثير من الملفات الصغيرة، ويطلق على مثل تلك الأدوات اسم المجمِّعات bundlers.

يتحكم حجم الملفات في سرعة نقلها عبر الشبكة كذلك، فليس العدد وحده هو المعيار، وعليه فقد اخترع مجتمع جافاسكربت مصغِّرات minifiers، وهي أدوات تأخذ برنامج جافاسكربت وتجعله أصغر عبر حذف التعليقات والمسافات تلقائيًا، كما تعيد تسمية الرابطات bindings، وتستبدل شيفرات أصغر بالشيفرات التي تأخذ حجمًا كبيرًا؛ وهكذا فليس من الغريب إيجاد شيفرة في حزمة NPM أو شيفرة تعمل في صفحة ويب، وتكون قد مرت بعدة مراحل من التحويل من جافاسكربت الحديثة إلى نسخة أقدم، ومن صيغة وحدات ES إلى CommonJS، ثم جُمِّعت وصغِّرت كذلك، كما لن نخوض في تفاصيل تلك الأدوات في هذا الكتاب لأنها مملة وتتغير بسرعة، لكن من المهم معرفة أنّ شيفرة جافاسكربت التي تشغِّلها لا تكون بهيئتها التي كُتبت بها أول مرة.

10.8 تصميم الوحدة

تُعَدّ هيكلة البرامج واحدةً من أدق الأمور في البرمجة، إذ يمكن نمذجة أيّ وظيفة غريبة بعدة طرق، كما يُعَدّ القول بجودة تصميم برنامج ما أمرًا نسبيًا، فهناك حلول وسط دومًا، كما تتدخل التفضيلات الشخصية في الحكم على التصميم، وأفضل ما يمكن فعله لتعلُّم قيمة التصميم الجيد هو القراءة والعمل على برامج كثيرة، وملاحظة ما ينجح وما يفشل، وإذا رأيت شيفرات فوضوية فلا تركن إلى قولك أن هذا هو الواقع ولا مجال لتغييره، بل هناك مساحة لتطوير هيكل أيّ شيء تقريبًا بإعمال الفكر فيه.

تُعَدّ سهولة الاستخدام أحد أركان تصميم الوحدات، فإذا كنت تصمم شيئًا ليستخدمه أشخاص عدة -أو حتى نفسك فقط-، فيُستحسن أن تكون واجهتك بسيطةً وسهلة الفهم والتوقع حين تنظر إليها بعد ثلاثة أشهر حين تنسى تفاصيل ما كتبته وما عملته، وقد يعني ذلك اتباع سلوكيات موجودة سلفًا، كما تُعَدّ حزمة `ini` مثالًا على ذلك، إذ تحاكي كائن `JSON` القياسي من خلال توفير دوال `parse` و `stringify` -لكتابة ملف INI- كما تحوِّل بين السلاسل النصية والكائنات المجردة مثل `JSON`، وهكذا فإنّ الواجهة صغيرة ومألوفة، وستذكر كيف استخدمتها لمجرد عملك معها مرةً واحدةً.

حتى لو لم تكن ثمة دالة قياسية أو حِزمة مستخدَمة على نطاق كبير لمحاكاتها، فستستطيع إبقاء وحداتك مألوفةً وسهلة التوقع باستخدام هياكل بيانات بسيطة تفعل أمرًا واحدًا فقط، كما توجد على NPM مثلًا وحدات عديدة لتحليل ملف-INI، إذ تحتوي على دالة تقرأ مثل هذا الملف من القرص الصلب مباشرة وتُحلله، ويجعل ذلك من المستحيل استخدام مثل تلك الوحدات في المتصفح، حيث لا يكون لنا وصولًا مباشرًا إلى نظام الملفات، كما يضيف تعقيدًا كان يمكن معالجته بأسلوب أفضل عبر تزويد composing الوحدة بدالة قارئة للملفات file-reading.

يشير ذلك إلى منظور آخر مفيد في تصميم الوحدات، وهو سهولة تطعيم شيء ما بشيفرة خارجية، إذ تكون الوحدات المركزة التي تحسب قيمًا قابلةً للتطبيق في طيف واسع من البرامج، على عكس الوحدات الأكبر التي تنفِّذ إجراءات معقدة لها آثار جانبية، وقارئ ملف INI الذي يركز على قراءة الملف من القرص الصلب ليس له

فائدة في سيناريو يكون فيه محتوى الملف من مصدر خارجي، وبالمثل فإنّ الكائنات الحالـة stateful objects مفيدة أحيانًا، بل قد تكون ضرورية؛ لكن إذا كان يمكن تنفيذ أمر ما بدالة فمن الأفضل استخدام الدالة.

توفِّر العديد من قارئات ملفات INI على NPM نمط واجهة interface style، إذ يحتـاج منـك إنشـاء كـائن أولً، ثم تحمِّل الملف إلى الكائن، وبعد ذلك تستخدِم توابع مخصصة للحصول على النتائج، وذلك الأمر شائع في البرمجة كائنية التوجه، كما هو أمر مرهق وخاطئ.

علينا تنفيذ طقوس نقل الكائن خلال عدة مراحل بدلًا من تنفيذ استدعاء واحد لدالة، وبما أن البيانات مغلفة الآن في نوع كائن مخصص، فيجب على كل الشيفرات التي تتفاعل معها معرفة ذلـك النـوع، ممـا يجعـل لـدينا اعتماديات غير ضرورية. قد تكون لدينا حالات لا يمكن تجنب تعريف هياكل بيانـات جديـدة فيهـا، حيث لا تـوفر اللغة إلا بعض الهياكل البسيطة، كما ستكون العديد من أنواع البيانات معقدةً أكثر من مجرد مصفوفة أو خارطـة map، لكن إذا كانت المصفوفة تكفي، فسنستخدِم المصفوفة.

يمكن النظر إلى مثال المخطط الذي تعرضنا لـه في الفصـل السـابع على أسـاس مثـال على هيكـل بيانـات معقد، إذ لا توجد طريقة واضحة لتمثيل المخطط في جافاسكربت، وقد استخدمنا في ذلك الفصل كائنًـا تحمـل خصائصه مصفوفات من السلاسل النصية، وهي العقد الأخرى التي يمكن الوصول إليها من تلك العقدة.

توجد العديد من حِزم إيجاد المسار في NPM، لكن لا تستخدِم أيّ منها صـيغة المخطـط تلـك، فهي تسـمح عادةً لحدود المخطط أن يكون لها وزن يكون التكلفة أو المسافة المرتبطة به، ولم يكن ذلك ممكنًا في تمثيلنا.

هناك حِزمة `dijkstrajs` مثلًا التي تستخدِم منظورًا شائعًا جدًا لإيجاد المسارات والذي يُسمى بخوارزميـة ديكسترا Dijkstra's algorithm، إذ سُمي باسم إدزجر ديكسترا Edsger Dijkstra الذي كتبه، وهو مشابه لدالة `findRoute` الخاصة بنا، ومن الشائع أن تضاف اللاحقة js إلى اسم الحِزمة لتوضيح أنها مكتوبة بجافاسكربت.

تستخدِم حزمة `dijkstrajs` صيغة مخطط تشبه صيغتنا، لكنهـا تسـتخدم كائنـات تكـون قيم خصائصـها أعدادًا -أي أوزان الحدود- بدلًا من المصفوفات التي استخدمناها، فإذا أردنا استخدام تلك الحزمة، فسيكون علينا التأكد من تخزين المخطط بالصيغة التي تتوقعها الحزمة، كما يجب أن تحصل جميع الحدود على الوزن نفسه بما أنّ نموذجنا المبسط يعامِل كل طريق على أساس امتلاكه التكلفة نفسها -أي منعطف واحد-.

```
const {find_path} = require("dijkstrajs");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
```

```
}

console.log(find_path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Salma's House", "Cabin"]
```

قد يكون ذلك حاجزًا معيقًا عن التركيب compositing، حيث تستخدِم حِزم عدة هياكل بيانات مختلفة لوصف أشياء متشابهة، فيكون جمعها معًا صعبًا، وبالتالي إذا أردنا تصميم شيء ليكون قابلًا للتركيب، فيجب علينا معرفة هياكل البيانات التي يستخدِمها الأشخاص أولًا، ثم نحذو حذوهم كلما تيسر.

10.9 خاتمة

توفر الوحدات هيكلًا لبرامج أكبر من خلال فصل الشيفرة إلى أجزاء صغيرة لها واجهات واضحة واعتماديات، كما تُعَدّ الواجهة الجزء المرئي من الوحدة للوحدات الأخرى، والاعتماديات هي الوحدات الأخرى التي تستخدِمها. ولأنّ جافاسكربت لم توفر نظامًا للوحدات في البداية، فقد بُني نظام CommonJS عليها، ثم حصلت جافاسكربت بعد مدة على نظام وحدات خاص بها، وهو الآن موجود إلى جانب CommonJS.

تُعَدّ الحزمة مجموعة شيفرات يمكن توزيعها مستقِلة بذاتها، كما يُعَدّ NPM مستودعًا لحِزم جافاسكربت، ونستطيع تحميل شتى الحِزم منه سواء مفيدة أو غير مفيدة.

10.10 تدريبات

10.10.1 الروبوت التركيبي

فيما يلي الرابطات التي ينشئها المشروع الذي في الفصل السابع:

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

إذا أردت كتابة هذا المشروع على أساس برنامج تركيبي modular، فما الوحدات التي ستنشئها؟ وما الوحدات التي ستعتمد عليها كل وحدة؟ وكيف ستبدو واجهاتها؟ وأيّ الأجزاء ستكون متاحةً ومكتوبةً مسبقًا على NPM؟ وهل ستفضِّل استخدام حِزمة NPM أم تكتبها بنفسك؟

إرشادات الحل

إليك ما كنا لنفعله بأنفسنا، لكن مرةً أخرى ليست هناك طريقة صحيحة وحيدة لتصميم وحدة معطاة:

توجد الشيفرة المستخدَمة لبناء مخطط الطريق في وحدة `graph`، ولأننا نُفضِّل استخدام `dijkstrajs` من NPM بدلًا من شيفرة إيجاد المسار التي كتبناها، فسنجعل هذا المخطط قريبًا من الذي تتوقعه `dijkstrajs`.

تصدِّر هذه الوحدة دالةً واحدةً هي `buildGraph`، كما سنجعل هذه الدالة تقبل مصفوفةً من مصفوفات ثنائية العناصر بدلًا من سلاسل نصية تحتوي على شخطات `-`، وذلك من أجل تقليل اعتماد الوحدة على صيغة الإدخال.

تحتوي وحدة `roads` على بيانات الطريق الخام -أي مصفوفة `roads`- ورابطة `roadGraph`، كما تعتمد تلك الوحدة على `./graph`. وتصدر مخطط الطريق.

يوجد صنف `VillageState` في وحدة `state`، حيث يعتمد على وحدة `./roads`. لأنه يحتاج إلى أن يكون قادرًا على التحقق من وجود طريق موجود فعليًا، كما يحتاج إلى `randomPick`، وبما أنّ هذه دالة من ثلاثة أسطر، فسنضعها في وحدة `state` على أساس دالة مساعدة داخلية، لكن `randomRobot` يحتاج إليها كذلك، لذا يجب تكرارها أو وضعها في وحدتها الخاصة، وبما أنّ تلك الدالة موجودة في NPM في حِزمة `random-item`، فمن الأفضل جعل كلا الوحدتين تعتمدان عليها، كما نستطيع إضافة دالة `runRobot` إلى تلك الوحدة أيضًا، بما أنها صغيرة ومرتبطة ارتباطًا وثيقًا بإدارة الحالة، في حين تصدِّر الوحدة كلا من الصنف `VillageStat` ودالة `runRobot`.

أخيرًا، يمكن دخول الروبوتات والقيم التي تعتمد عليها مثل `mailRoute` في وحدة `example-robots` التي تعتمد على `./roads`. وتصدِّر دوال الروبوت، ولكي يستطيع `goalOrientedRobot` البحث عن المسار، فستعتمد الوحدة على `dijkstrajs` أيضًا.

صارت الشيفرة أصغر قليلًا من خلال نقل بعض المهام إلى وحدات NPM، حيث تنفِّذ كل وحدة مستقِلة شيئًا بسيطًا ويمكن قراءتها بنفسها، ويقترح تقسيم الشيفرات إلى وحدات على أساس تحسينات أكثر لتصميم البرنامج.

يُعد اعتماد الروبوتات و`VillageState` على مخطط طريق بعينه غريبًا بعض الشيء، فربما يكون من الأفضل جعل المخطط وسيطًا لباني الحالة، ونجعل الروبوتات تقرؤه من كائن الحالة، فنقلل الاعتماديات -وهو أمر جيد-، ونجعل من الممكن إجراء عمليات محاكاة على خرائط مختلفة، وذلك خير وأفضل.

هل من الجيد استخدام وحدات NPM لأمور كان يمكن كتابتها بأنفسنا؟ نظريًا، نعم، فمن المرجح أنك ستقع في أخطاء في الأمور المعقَّدة مثل دالة إيجاد المسار وتضيع وقتك في كتابتها بنفسك؛ أما بالنسبة للدوال الصغيرة مثل `random-item` فتكون كتابتها أمرًا يسيرًا، لكن إضافتها في أيّ مكان تحتاج إليها فيه سيعكر وحداتك.

لكن بأي حال، لا تقلل من شأن الجهد المطلوب لإيجاد حِزمة NPM مناسبة، فحتى لو وجدتها فقد لا تعمل جيدًا أو تكون مفتقدةً إلى بعض المزايا التي تحتاج إليها، وفوق هذا يعني الاعتماد على حِزم NPM التأكد من أنها مثبَّتة، كما ستنشرها مع برنامجك، بل ربما يكون عليك ترقيتها دوريًا، فهذه قد تكون إحدى التبعات التي عليك تحملها، وعلى أيّ حال تستطيع اتخاذ القرار الذي يناسبك وفقًا لمقدار العون الذي تقدمه تلك الحزم لك.

10.10.2 وحدة الطرق

اكتب وحدة CommonJS بناءً على المثال الذي في الفصل السابع، بحيث تحتوي على مصفوفة من الطرق وتصدِّر هيكل بيانات المخطط الذي يمثلها على أساس `roadgraph`، كما يجب أن تعتمد على وحدة `./graph` التي تصدِّر الدالة `buildGraph` المستخدَمة لبناء المخطط، وتتوقع هذه الدالة مصفوفةً من مصفوفات ثنائية العناصر -أي نقاط البداية والنهاية للطرق-.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// أضف اعتماديات وتصديرات.
const roads = [
  "Salma's House-Omar's House",   "Salma's House-Cabin",
  "Salma's House-Post Office",   "Omar's House-Town Hall",
  "Sara's House-Mostafa's House", "Sara's House-Town Hall",
  "Mostafa's House-Sama's House", "Sama's House-Farm",
  "Sama's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```

إرشادات الحل

بما أنّ هذه وحدة CommonJS، فعليك استخدام `require` لاستيراد وحدة المخطط، وقد وُصف ذلك على أساس تصدير دالة `buildGraph`، إذ تستطيع استخراجه من كائن واجهته مع تصريح فك الهيكلة `const`.

أضف خاصيةً إلى كائن `exports` من أجل تصدير `roadGraph`، كما يجب أن يكون فصل سلاسل الطريق النصية داخل وحدتك لأن `buildGraph` تأخذ هيكل بيانات لا يطابق `roads` بالضبط.

10.10.3 الاعتماد المتبادل بين الوحدات

تُعَدّ حالة الاعتماد المتبادل بين وحدتين ويدعى circular dependency حالةً تعتمد فيها الوحدة A على الوحدة B -أي يكون الاعتماد على شكل حلقة تبادلية بين وحدتين ولذلك سمي باللغة الإنجليزية circular dependency-، والعكس صحيح مباشرةً أو غير مباشرة، كما تمنع العديد من أنظمة الوحدات ذلك بسبب استطاعتك التأكد من تحميل اعتماديات وحدة قبل تشغيلها، بعض النظر عن الترتيب الذي تختاره لتحميل مثل تلك الوحدات.

تسمح وحدات CommonJS بصورة محدودة من الاعتماديات الدورية cyclic dependencies تلك، طالما أنّ الوحدات لا تستبدل كائن `exports` الافتراضي الخاص بها، ولا يكون لها وصول إلى واجهة غيرها حتى تنتهي من التحميل.

تدعم دالة `require` التي تعرضنا لها سابقًا في هذا الفصل مثل ذلك النوع من تبادلية الاعتماديات dependency cycle، فهل يمكنك رؤية كيف تعالج هذه الحالة؟ وما الذي قد يحدث إذا استبدلت وحدة في تعتمد وحدة أخرى عليها (أي داخلة في دورة الاعتمادية)دورة، كائن `exports` الافتراضي الخاص بها؟

إرشادات الحل

تدور الفكرة هنا حول إضافة `require` وحدات إلى ذاكرتها المؤقتة قبل بدء تحميل الوحدة، وهكذا فإذا حاول أيّ استدعاء `require` تحميلها أثناء تشغيلها، فسيكون معروفًا وتُعاد الواجهة الحالية بدلًا من تحميل الوحدة مرةً أخرى، وذلك يتسبب في طفحان المكدس؛ أما إذا أعادت وحدة كتابة قيمة `module.exports` الخاصة بها، فستحصل أيّ وحدة أخرى كانت قد استقبلت قيمة واجهتها قبل أن تنهي التحميل، على كائن الواجهة الافتراضي -الذي سيكون فارغًا على الأرجح- بدلًا من قيمة الواجهة المقصودة.

دورة إدارة تطوير المنتجات

تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق
وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن

11. البرمجة غيـر المتـزامنة

قد يدرك المتأني بعض حاجته، وقد يكون مع المستعجل الزلل.

— حكمة شهيرة

معالج الحاسوب هو تلك الرقاقة المركزيـة بين دارات الحاسـوب المنفِّذة للخطـوات الصـغيرة الـتي تتشـكل منها البرامج، وما كانت البرامج التي رأيناها حتى الآن إلا أشياء تبقيه مشـغولًا إلى أن تنهي مهامهـا، كمـا تعتمـد سرعة تنفيذ تعليمة برمجية مثل حلقة تكرارية تعالج بيانات على سرعة المعالج نفسه.

تتواجد مع هذا عدة برامج تتفاعل مع أشـياء خـارج المعـالج، فقـد تتواصـل عـبر شـبكة حاسـوبية، أو تطلب بيانات من قرص صلب مثلًا، والذي قد يكون أبطأ من الحصول على المعلومة نفسها من القرص الصلب، وحين يحدث مثل ذلك، فعلينا معرفة ما إذا كان يليق بنا ترك المعالج يقبع ساكنًا دون عمل، ولهذا لا بد من وجود عمل يمكنه تنفيذه في ذلك الوقت، كما يعالج نظام تشغيلك ذلك جزئيًا، حيث سيجعل المعالج يتنقل بين عدة بـرامج عاملة، لكن هذا لا ينفع حين نريد لبرنامج واحد العمل بينما ينتظر ردًا من الشبكة.

11.1 عدم التزامن Asynchronicity

تحدث الأشياء في نموذج البرمجة المتزامنة synchronous واحـدةً تلـو الأخـرى، فلا تعيـد دالـة تنفِّذ إجـراءً يعمل على المدى الطويل إلا إذا وقع الإجراء وصار بإمكانه إعادة النتيجة، وبالتالي سيُوقِف هذا برنامجك إلى حين تنفيذ ذلك الإجراء؛ أما النموذج غير المتزامن asynchronous فسيسمح بحدوث عدة أشياء في الـوقت نفسـه، حيث سيستمر برنامجـك بالعمـل إذا بـدأت بـإجراء مـا، كمـا سـيُبلَّغ البرنـامج حين ينتهي الإجـراء ويحصـل على وصول access إلى النتيجة مثل قراءة البيانات من القرص.

نستطيع موازنة البرمجة المتزامنة وغير المتزامنة باستخدام مثال بسيط، وهو برنامج يجلب مَوردَين من الشبكة ثم يوازن النتيجة، بحيث تُعيد دالة الطلب في البيئة المتزامنة بعد إنهاء عملها. وأسهل طريقة لتنفيذ ذلك الإجراء هي جعل الطلبات واحدًا تلو الآخر، وهذا له تبعته، إذ لن يبدأ الطلب الثاني إلا حين ينتهي الطلب الأول، كما سيكون الوقت الكلي المستغرق مساويًا وقت رد الطلبَين معًا.

يكون حل تلك المشكلة في النظام المتزامن ببدء خيوط تحكم control threads إضافية، وقد يكون الخيط هنا برنامجًا آخرًا عاملًا قد يتزامن تنفيذه مع برامج أخرى بواسطة نظام التشغيل، وبما أنّ أغلب الحواسيب الحديثة تحتوي على عِدة معالجات، فقد تعمل الخيوط المتعددة في الوقت نفسه، بل قد تعمل على معالجات مختلفة، فيبدأ خيط جديد الطلب الثاني، ثم ينتظر الخَيطان عودة نتائجهما، وبعد ذلك يُعيدا المزامنة من أجل دمج النتائج.

تمثِّل الخطوط السميكة thick lines في المخطط التالي الوقت الذي ينفقه البرنامج في العمل العادي، في حين تمثِّل الخطوط الرفيعة thin lines وقت انتظار الشبكة، كما يكون الوقت الذي تأخذه الشبكة في النموذج المتزامن جزءًا من الخط الزمني timeline لخط تحكم ما؛ أما في النموذج غير المتزامن، فيتسبب بدء إجراء شبكة في حدوث انقسام split في الخط الزمني، كما يستمر البرنامج الذي ابتدأ الحدث بالعمل، ويقع الحدث معه أيضًا، ثم ينبه البرنامج حين ينتهي.

هناك طريقة أخرى تصف الفرق بأنّ انتظار انتهاء الأحداث في النموذج المتزامن أمر ضمني implicit، بينما يكون صريحًا explicit وتحت تحكمنا في النموذج غير المتزامن.

تتقاطع اللاتزامنية مع كلا الطريقتين، فهي تسهل التعبير عن البرامج التي لا تتوافق مع النموذج الخطي للتحكم، في حين قد تصعِّب التعبير عن البرامج التي تتبع النموذج الخطي، وسنرى بعض الطرق للنظر في تلك الغرابة في هذا الفصل لاحقًا.

تجعل المنصَّتان الهامتان لجافاسكربت -أي المتصفحات وNode.js- العمليات التي تستغرق وقتًا غير متزامنة، بدلًا من الاعتماد على الخيوط، وذلك جيد بما أنّ البرمجة بالخيوط صعبة جدًا بسبب صعوبة فهم مهمة البرنامج حين ينفِّذ أمورًا كثيرةً في وقت واحد.

11.2 تقنية الغراب

تستطيع الغربان استخدام الأدوات والتخطيط وتذكر الأشياء، وإيصال هذه الأشياء فيما بينهم، إذ أن شأنها في ذلك شأن غيرها من المخلوقات.

لكن ما لا يعلمه الكثير عنها أنها قادرة على أمور أكثر من ذلك، إذ تُنشئ العديد من أسراب الغربان أجهزةً حاسوبيةً حيويةً مثلًا، فهي ليست إلكترونية مثل التي نصنعها، لكنها تتكون من حشرات دقيقة أشبه بالنمل الأبيض، إذ تُطوِّر معها علاقةً تكافليةً بحيث تزودها الغربان بالطعام مقابل أن تبني الحشرات تلك المستعمرات المعقدة وتشغلها، لتُجري حسابات لاحقًا بمساعدة كائنات حية داخلها.

تقع مثل تلك المستعمرات عادةً في أعشاش كبيرة، حيث تعمل الطيور والحشرات معًا فيها لبناء هياكل طينية بصلية الشكل مخفية داخل أغصان الأعشاش تعيش فيها الحشرات وتعمل، وتَستخدِم تلك الآلات إشارات ضوئيةً للتواصل مع الأجهزة الأخرى، إذ تُدخِل الغربان قطعًا عاكسةً للضوء في سيقان خاصة بالتواصل، ثم توجه الحشرات تلك السيقان لعكس الضوء للأعشاش الأخرى، وهذا يعني أنّ الأعشاش التي بينها مجال بصري مفتوح هي التي تستطيع التواصل فقط.

رسم أحد الخبراء خارطةً لشبكة أعشاش الغربان في قرية إييغ-سوغ-امبي Hières-sur-Amby الفرنسية، بحيث توضِّح أماكن الأعشاش واتصالاتها، كما في الشكل التالي:

سنكتب في هذا الفصل بعض الدوال الشبكية البسيطة لمساعدة الغربان على التواصل.

11.3 ردود النداء Callbacks

يمكن النظر إلى البرمجة غير المتزامنة على أنها تجعل الدوال التي تنفِّذ إجراءً بطيئًا تأخذ وسيطًا إضافيًا يكون دالة رد نداء callback function، حيث يبدأ الإجراء، ثم تُستدعى دالة رد النداء بالنتيجة حين ينتهي.

لدينا مثلًا دالة `setTimeout` المتاحة في المتصفحات وNode.js على سواء، إذ تنتظر زمنًا بعينه يقاس بالميلي ثانية، ثم تَستدعي الدالة.

```
setTimeout(() => console.log("Tick"), 500);
```

لا يُعَد الانتظار عملًا مهمًا، لكن قد يكون مفيدًا حين نفعل أمرًا مثل تحديث رسم تحريكي أو التحقق من استغراق شيء وقتًا أطول من الوقت المسموح به، ويعني تنفيذ عدة أحداث غير متزامنة متتالية باستخدام ردود النداء؛ أنه علينا الاستمرار بتمرير دوال جديدة لمعالجة استمرارية الحوسبة بعد الحدث.

تحتوي أغلب حواسيب أعشاش الغربان على حوصلة bulb تخزين بيانات طويل الأمد، حيث تُحفَر أجزاء المعلومات في الأغصان كي تُستخدَم لاحقًا، ويستغرق ذلك الحفر أو إيجاد جزء من البيانات وقتًا، لذا فإن واجهة التخزين طويل الأمد غير متزامنة، كما تَستخدِم دوال رد نداء.

تخزِّن حواصل التخزين أجزاءً من بيانات JSON القابلة للتشفير تحت أسماء، وقد يخزِّن الغراب معلومات عن المكان الذي فيه الطعام المخبأ تحت اسم `"food caches"`، والذي قد يحمل مصفوفةً من الأسماء التي تشير إلى أجزاء أخرى من البيانات التي تصف الذاكرة المؤقتة الحقيقية، كما سيشغل الغراب مثل الشيفرة التالية للبحث عن مخزون طعام في حواصل التخزين storage bulbs في عش "Big Oak":

```
import {bigOak} from "./crow-tech";

bigOak.readStorage("food caches", caches => {
  let firstCache = caches[0];
  bigOak.readStorage(firstCache, info => {
    console.log(info);
  });
});
```

يُعَدّ هذا النسق من البرمجة قابلًا للتطبيق، لكنه يزيد من مستوى الإزاحة مع كل حدث غير متزامن، وذلك لأننا سنصل إلى دالة أخرى تفعل أمورًا أعقد من التي بين أيدينا لا محالة، مثل تنفيذ أحداث متعددة في الوقت نفسه، إذ سيكون الأمر غير مألوف.

تُبنى حواسيب الغربان العشيَّة لتتواصل فيما بينها باستخدام أزواج من طلب-رد request-response، وهذا يعني أنّ أحد الأعشاش سيرسل رسالةً إلى عش آخر، ثم يرد الثاني مباشرةً مؤكدًا استلام الرسالة، وربما يرد

أيضًا على الرسالة في الوقت نفسه، كما تُوسَم كل رسالة بنوع يحدد كيفية معالجتها، إذ تستطيع شيفرتنا تعريف المعالِجات handlers لكل نوع من أنواع الطلبات، ويُستدعى المعالج متى ما دخل ذلك الطلب لإنتاج رد.

توفِّر الواجهة التي صدَّرتها وحدة `"./crow-tech"` دوالًا مبنيةً على ردود النداء من أجل التواصل، كما تحتوي الأعشاش على التابع `send` الذي يرسل الطلب، إذ يتوقع اسم العش الهدف ونوع الطلب ومحتواه على أساس أول ثلاثة وسائط، ثم يتوقع دالةً لتُستدعى حين يأتي الرد على أساس وسيط رابع وأخير.

```
bigOak.send("Cow Pasture", "note", "Let's caw loudly at 7PM",
            () => console.log("Note delivered."));
```

لكن يجب تعريف نوع طلب أولًا باسم `"note"` لتمكين الأعشاش من استقبال ذلك الطلب، كما يجب أن تعمل الشيفرة التي تعالج الطلبات على جميع الأعشاش التي تستطيع استقبال رسالة من هذا النوع وليس على حاسوب العش فقط، حيث سنفترض أنّ الغراب سيطير هنا ليثبِّت شيفرة المعالج تلك على جميع الأعشاش.

```
import {defineRequestType} from "./crow-tech";

defineRequestType("note", (nest, content, source, done) => {
  console.log(`${nest.name} received note: ${content}`);
  done();
});
```

تُعرِّف الدالة `defineRequestType` نوعًا جديدًا من الطلبات، كما يضيف المثال الدعم لطلبات `"note"` التي ترسِل تذكرة note إلى العش المعطى، ويَستدعي تطبيقنا `console.log` كي نستطيع توكيد وصول الطلب، كما تحتوي الأعشاش على الخاصية `name` التي تحمل أسماءها؛ أما الوسيط الرابع المعطى للمعالج فهو `done`، وهي دالة رد نداء يجب أن تستدعي عند انتهاء الطلب، فإذا استخدمنا قيمة الإعادة للمعالج على أساس قيمة رد، فهذا يعني عدم استطاعة معالج الطلب تنفيذ إجراءات غير متزامنة بنفسه، فالدالة التي تنفذ مهامًا غير متزامنة تُعيد قبل انتهاء العمل، كما تكون قد جهزت رد نداء عند انتهاءها، لذا فنحن نحتاج إلى آلية غير متزامنة، وهي دالة رد نداء أخرى في حالتنا لترسل إشعارًا حين يتاح ردًا.

قد يكون اللاتزامن معديًا contagious نوعًا ما، فأيّ دالة تستدعي دالةً أخرى تعمل بأسلوب غير متزامن يجب أن تكون هي نفسها غير متزامنة، ومستخدمةً لرد نداء أو آليةً مشابهةً لتسليم نتيجتها؛ أما استدعاء رد النداء ففيه تفصيل أكثر من مجرد إعادة قيمة ما، كما يكون عرضةً للأخطاء، لذا فإنّ الحاجة إلى هيكلة أجزاء كبيرة من برنامجنا بهذه الطريقة ليست الأسلوب الأمثل.

11.4 الوعود

يسهل العمل مع المفاهيم المجردة حين يمكن تمثيلها بقيم، وفي حالة الإجراءات غير المتزامنة سنستطيع إعادة كائن يمثل الحدث المستقبلي، بدلًا من إعداد دالة لتُستدعى في نقطة ما في المستقبل، وهذه هي وظيفة الصنف `Promise`، فالوعد هو إجراء غير متزامن قد يحدث عند نقطة ما ويعطينا قيمة، ويستطيع إبلاغ أيّ أحد يريد إشعاره حين تكون القيمة متاحةً.

يُعَدّ استدعاء `Promise.resolve` أسهل طريقة لإنشاء وعد، وهي دالة تضمن أنّ القيمة التي تعطيها إياها ستغلَّف داخل وعد، فإذا كان هي نفسها وعدًا، فستعاد ببساطة، وإلا فستحصل على وعد ينتهي مباشرةً بقيمتك على أساس نتيجة.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

نستطيع استخدام التابع `then` للحصول على وعد، إذ يسجل دالة رد نداء لتُستدعى حين ينتهي الوعد وينتج قيمةً، كما نستطيع إضافة عدة ردود نداء إلى وعد واحد، إذ ستُستدعى جميعًا حتى لو أضفتها بعد انتهاء الوعد، ومع هذا فلا زال التابع `then` يحتوي مزيدًا من الخصائص، إذ يُعيد وعدًا آخرًا ينتهي بالقيمة التي تعيدها دالة المعالِج، أو إذا أعاد ذلك وعدًا، فسينتظر ذلك الوعد وينتهي مُخرجًا نتيجته.

يُعَدّ النظر للوعود على أنها جهاز لنقل القيم إلى بيئة غير متزامنة مفيدًا، فالقيمة العادية موجودة والقيمة الموعودة promised value قد تكون موجودة، كما قد تظهر في نقطة ما في المستقبل، كما تتصرف الحسابات المعرَّفة بالوعود على مثل هذه القيم المغلَّفة wrapped values وتنفَّذ تنفيذًا غير متزامن عندما تكون القيم متاحة.

نستطيع استخدام `Promise` على أساس باني لكي ننشئ وعدًا، لكن هذا سيكون له واجهة غريبة نوعًا ما، إذ يتوقع الباني دالةً على أساس وسيط ويستدعيها مباشرةً، ثم يمرر إليها دالةً تستطيع استخدامها لحل الوعد وإنهائه، وسبب عملها بهذا الأسلوب بدلًا من استخدام التابع `resolve` مثلًا، هو اقتصار حل الوعد على الشيفرة التي أنشأته.

يوضِّح المثال التالي كيفية إنشاء واجهة مبنية على وعد لدالة `readStorage`:

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}
```

```
storage(bigOak, "enemies")
  .then(value => console.log("Got", value));
```

تُعيد هذه الدالة غير المتزامنة قيمةً ذات معنى، وهذه هي الميزة الأساسية للوعود، إذ تبسِّط استخدام الدوال غير المتزامنة، فبدلًا من الحاجة إلى تمرير ردود النداء كما تبدو الدوال المبنية على الوعود أشبه بالدوال العادية؛ فهي تأخذ الدخل على أساس وسائط وتُعيد مخرجاتها، ولا تختلف إلا أنّ الخرج قد يكون غير متاح بعد.

11.5 الفشل Failure

يمكن أن تفشل حسابات جافاسكربت العادية برفع استثناء exception، إذ تحتاج الحوسبة غير المتزامنة إلى ذلك عادةً، فقد يفشل طلب الشبكة أو شيفرة ترفع استثناءً وتكون جزءًا من الحوسبة غير المتزامنة.

إحدى المشاكل المؤرقة في نمط رد النداء للبرمجة غير المتزامنة أنها تصعِّب ضمان الإبلاغ الصحيح لردود النداء بشأن حالات الفشل، وأحد الحلول المستخدَمة على نطاق واسع هو استخدام وسيط رد النداء الأول لتوضيح فشل الإجراء، بعدها سيحتوي الوسيط الثاني على القيمة التي أنتجها الإجراء حين كان ناجحًا.

يجب أن تتحقق دوال ردود النداء هذه إذا كانت قد استقبلت استثناءً أم لا، كما عليها التأكد من أنّ أيّ مشكلة تسببها بما فيها الاستثناءات المرفوعة من قِبَل الدوال التي استدعتها، قد عُلم بها وسُلِّمت إلى الدالة المناسبة؛ أما الوعود فتيسِّر من هذه العملية كثيرًا، فهي إما محلولة -أي انتهى الإجراء بنجاح-، أو مرفوضة -أي فشلت-، إذ لا تُستدعى معالِجات الإنهاء resolve handlers التي سُجلت في `then` إلا عند نجاح الإجراء؛ أما عمليات الرفض فتُنقَل إلى الوعد الجديد الذي تُعيده `then`. وحين يرفع معالج استثناءً، فسيجعل هذا الوعد الذي أنتجه الاستدعاء إلى `then` مرفوضًا.

لهذا إذا فشل أيّ عنصر في سلسلة إجراءات غير متزامنة، فسيحدَّد الناتج الكلي للسلسلة على أنه مرفوض، ولا تُستدعى معالِجات نجاح أبعد من النقطة التي فشلت فيها.

كذلك فإنّ رَفْض وعد ما ينتج قيمةً تمامًا مثل التي ينتجها حل الوعد، كما يُطلق على تلك القيمة سبب الرفض، فإذا تسبب استثناء في دالة معالِجة بالرفض، فستُستخدِم قيمة الاستثناء على أساس سبب، وبالمثل، إذا أعاد معالج وعدًا مرفوضًا، فسينتقل ذلك الرفض إلى الوعد التالي، حيث لدينا دالة `Promise.reject` التي تنشئ وعدًا جديدًا مرفوضًا فورًا.

تحتوي الوعود على التابع `catch` لمعالجة عمليات الرفض تلك صراحةً، إذ يسجِّل معالجًا ليُستدعى حين يُرفض الوعد مثلما تعالج معالجات `then` الحل العادي للوعود، كما تشبه `then` في أنها تُعيد وعدًا جديدًا، إذ يحل إلى قيمة الوعد الأصلية إذا حل حلًا عاديًا، وإلى نتيجة معالج `catch` في غير ذلك.

يُرفَض الوعد الجديد كذلك إذا رفع معالج catch خطأً ما، وللاختصار، تقبل then معالج الرفض على أساس وسيط ثاني، لذا نستطيع تثبيت كلا النوعين من المعالجات في استدعاء تابع وحيد، كما تستقبل الدالة الممرَّرة إلى الباني Promise وسيطًا ثانيًا مع دالة حل resolve function لتستخدمها في رفض الوعد الجديد.

يمكن النظر إلى سلاسل قيم الوعود المنشأة باستدعاءات إلى then وcatch على أساس خط أنابيب تتحرك فيه عمليات الفشل أو القيم غير المتزامنة، وبما أنّ تلك السلاسل قد أُنشئت بتسجيل معالجات، فسيكون لكل رابط معالج نجاح أو معالج فشل أو كليهما؛ حيث تُهمَل المعالِجات التي لا تطابق نوع الخرج - بنجاح أو فشل-؛ أما المعالجات التي تطابق فستُستدعى ويحدِّد خرجها نوع القيمة التي ستأتي تاليًا، بحيث تكوّن نجاحًا حين تُعيد قيمةً ليست وعدًا، وفشلًا حين ترفع استثناءً، وخرْج الوعد حين تُعيد أحد هذين.

```
new Promise((_, reject) => reject(new Error("Fail")))
  .then(value => console.log("Handler 1"))
  .catch(reason => {
    console.log("Caught failure " + reason);
    return "nothing";
  })
  .then(value => console.log("Handler 2", value));
// → Caught failure Error: Fail
// → Handler 2 nothing
```

تُعالَج الاستثناءات غير الملتقَطة بواسطة البيئة، إذ تستطيع بيئات جافاسكربت استشعار إذا لم يعالَج رفض الوعد، وستُبلِّغ هذا في صورة خطأ.

11.6 صعوبة الشبكات

قد تأتي أيام لا يكون فيها ضوء كافي لنظام المرايا الذي يستخدمه الغربان من أجل نقل إشارة، أو قد يحجب شيء ما مسار الإشارة، كذلك قد تُرسَل إشارة ولا تُستقبَل من الطرف الآخر، حيث سيتسبب ذلك في عدم استدعاء رد النداء المعطى إلى send، وهو الأمر الذي سيجعل البرنامج يتوقف دون أي ملاحظة حتى لوجود مشكلة، فلو بلَّغ الطلب بالفشل إذا مر زمن محدد دون رد، لكان أفضل.

تكون حالات فشل الإرسال في العادة حوادث عرضية مثل تداخل الأضواء القادمة من مصابيح سيارة مع الإشارات الضوئية، وهنا سينجح الطلب بمجرد إعادة المحاولة، لذا سنجعل دالة الطلب تحاول تلقائيًا إعادة إرسال الطلب عدة مرات قبل أن تستسلم.

وبما أننا عرفنا الآن أن الوعود أمر جيد، فسنجعل دالة الطلب تُعيد وعدًا، إذ يتشابه رد النداء والوعد من حيث ما يستطيعان التعبير عنه، كما يمكن تغليف الدوال المبنية على ردود النداء لكشف واجهة مبنية على وعد، والعكس صحيح.

قد يشير الرد إلى فشل عند تسليم الطلب ورده بنجاح إذا حاول الطلب استخدام نوع طلب غير معرَّف أو رفع المعالج خطأ مثلًا.

لدعم هذا يتَّبع كل من `send` و`defineRequestType` الأسلوب المذكور أعلاه، حيث يكون الوسيط الأول الممرَّر إلى ردود النداء هو سبب الفشل إذا وُجد، في حين يكون الثاني هو النتيجة الحقيقية، ويمكن ترجمة هذا إلى حل الوعد ورفضه باستخدام المغلِّف الخاص بنا.

```
class Timeout extends Error {}

function request(nest, target, type, content) {
  return new Promise((resolve, reject) => {
    let done = false;
    function attempt(n) {
      nest.send(target, type, content, (failed, value) => {
        done = true;
        if (failed) reject(failed);
        else resolve(value);
      });
      setTimeout(() => {
        if (done) return;
        else if (n < 3) attempt(n + 1);
        else reject(new Timeout("Timed out"));
      }, 250);
    }
    attempt(1);
  });
}
```

سينجح ما فعلناه أعلاه لأنّ الوعود لا يمكن حلها أو رفضها إلا مرةً واحدة، إذ تُحدِّد المرة الأولى التي يُستدعى فيها `resolve` أو `reject` خرج الوعد، كما تُتجاهل الاستدعاءات التالية التي سببها طلب عائد بعد انتهاء طلب آخر.

سنحتاج إلى استخدام دالة تعاودية لبناء حلقة غير متزامنة من أجل إعادة المحاولة التي ذكرناها، حيث لا تسمح الحلقة العادية بالتوقف وانتظار الإجراء غير المتزامن، وتنفِّذ الدالة `attempt` محاولةً واحدةً لإرسال الطلب وتضبط زمن مهلة محدد، فإذا لم تحصل على استجابة أو رد بعد ربع ثانية، فستبدأ المحاولة الثانية أو ترفض الوعد إذا كانت هذه هي المحاولة الثالثة، ويكون السبب هنا هو نسخة من `Timeout`.

تُعَدّ المحاولة كل ربع ثانية وتركها إذا لم نتلق ردًا بعد ثلاثة أرباع ثانية نظامًا عشوائيًا نوعًا ما، فمن المحتمل إذا جاء الطلب لكن المعالج استغرق وقتًا أطول قليلًا أن تُسلَّم الطلبات أكثر من مرة، وسنكتب معالجاتنا واضعين تلك المشكلة في حساباتنا، فلا يجب أن تمثل الرسائل المكررة مشكلة.

لن نبني شبكة عالية المستوى هنا، فسقف توقعات الغربان ليس عاليًا على أي حال حين يتعلق الأمر بالحوسبة، ولكي نعزل أنفسنا من ردود النداء كلها، فسنعرِّف مغلِّفًا لـ `defineRequestType` يسمح لدالة المعالِج أن تُعيد وعدًا أو قيمةً مجردةً plain value ويوصل ذلك لرد النداء من أجلنا.

```
function requestType(name, handler) {
  defineRequestType(name, (nest, content, source,
                           callback) => {
    try {
      Promise.resolve(handler(nest, content, source))
        .then(response => callback(null, response),
              failure => callback(failure));
    } catch (exception) {
      callback(exception);
    }
  });
}
```

نستخدِم `Promise.resolve` لتحويل القيمة التي يُعيدها `handler` إلى وعد إذا لم تُحوَّل فعليًا.

لاحظ أنه يجب تغليف الاستدعاء إلى `handler` في كتلة `try` لضمان توصيل أي استثناءات يرفعها مباشرةً إلى رد النداء، حيث يوضح هذا صعوبة معالجة الأخطاء التي فيها ردود نداء معالجة صحيحة، ومن السهل نسيان كيفية توجيه استثناءات مثل هذه، وإذا لم نفعل ذلك فلن يُبلّغ بحالات الفشل إلى رد نداء مناسب؛ أما الوعود فستجعل هذا آليًا تقريبًا ومن ثم يكون أقل عرضةً للأخطاء.

11.7 تجميعات الوعود

يحتفظ حاسوب العش بمصفوفة من الأعشاش الأخرى التي في نطاق الاتصال في الخاصية `neighbors`.

سنكتب دالةً تحاول إرسال طلب `"ping"` إلى كل حاسوب لنرى أيها ترد علينا، وهو طلب يسأل ردًا ببساطة، وذلك لنرى إن كان مكن الوصول إلى أيّ من تلك الأعشاش.

تُعَدّ الدالة `Promise.all` مفيدةً عند العمل مع تجميعات من الوعود التي تعمل في نفس الوقت، إذ تُعيد وعدًا ينتظر حل جميع الوعود التي في المصفوفة، ثم يُحل هو إلى مصفوفة من القيم التي أنتجتها تلك الوعود بترتيب المصفوفة الأصلية نفسها، وإذا رُفض أي وعد فستُرفَض نتيجة `Promise.all` كذلك.

```
requestType("ping", () => "pong");

function availableNeighbors(nest) {
  let requests = nest.neighbors.map(neighbor => {
    return request(nest, neighbor, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(requests).then(result => {
    return nest.neighbors.filter((_, i) => result[i]);
  });
}
```

لا نريد فشل الوعد المجمَّع إذا كان أحد الجيران غير متاح ونحن ما زلنا لا نعرف باقي البيانات، لذا تُلحِق الدالة التي رُبطت بمجموعة من الجيران لتحولهم إلى وعود طلبات، حيث أن المعالجات تصنع طلبات ناجحة تنتج `true` وطلبات فاشلة تنتج `false`.

يُستخدَم `filter` في معالج الوعد المجمَّع لحذف العناصر من مصفوفة `neighbors` التي تكون قيمتها الموافقة هي `false`، إذ يستفيد هذا من كون `filter` يمرر فهرس مصفوفة العنصر الحالي على أساس وسيط ثاني إلى دالة الترشيح، كما يفعل `map` و`some` وتوابع المصفوفات العليا المشابهة.

11.8 إغراق الشبكة Network flooding

يحد اقتصار تكلم الأعشاش لجيرانها كثيرًا من فائدة هذه الشبكة، وعلى ذلك فربما نحاول إعداد نوع من الطلبات يُعيد توجيه الطلبات تلقائيًا إلى الجيران، وذلك على أساس حل لنشر المعلومات إلى الشبكة كلها، ثم يُعيد هؤلاء الجيران توجيهها إلى جيرانهم، وهكذا حتى تستقبل الشبكة كلها تلك الرسالة.

```
import {everywhere} from "./crow-tech";

everywhere(nest => {
  nest.state.gossip = [];
});

function sendGossip(nest, message, exceptFor = null) {
  nest.state.gossip.push(message);
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "gossip", message);
```

```
    }
  }

  requestType("gossip", (nest, message, source) => {
    if (nest.state.gossip.includes(message)) return;
    console.log(`${nest.name} received gossip '${
                 message}' from ${source}`);
    sendGossip(nest, message, source);
  });
```

يحتفظ كل عش بمصفوفة من سلاسل gossip النصية التي رآها فعليًا، وذلك لتجنب تكرار إرسال الرسالة عبر الشبكة للأبد، كما نستخدِم دالة `everywhere` لتعريف تلك المصفوفة، إذ تُشغِّل هذه الدالة الشيفرة على كل عش من أجل إضافة خاصية إلى كائن `state` الخاص بالعش، وهو المكان الذي سنحفظ فيه حالة العش المحلية.

يتجاهل العش رسالة gossip مكرَّرة مرسَلة إليه، الأمر الذي يحدث عادةً حين يعيد الناس إرسالها دون وعي، لكن حين يستلم رسالةً جديدةً، فسيخبر جميع جيرانه عدا ذلك الذي أرسلها.

سينشر هذا جزءًا جديدًا من gossip في الشبكة كما تنتشر بقعة الحبر في الماء، فحتى إذا لم تكن بعض الاتصالات عاملةً وقتها، فستصل gossip العش المعطى إذا كان ثمة طريق بديلة إليه.

يسمى ذلك الأسلوب في التواصل الشبكي باسم الإغراق أو الغمر flooding، فهو يغرق الشبكة بجزء من المعلومات إلى أن تكون لدى جميع العقد.

نستطيع استدعاء `sendGossip` لنرى تدفق الرسائل خلال القرية.

```
sendGossip(bigOak, "Kids with airgun in the park");
```

11.9 توجيه الرسائل

إذا أرادت عقدة ما التواصل مع عقدة أخرى فلن يكون أسلوب الإغراق هنا هو الأمثل، خاصةً حين تكون الشبكة كبيرة، فقد يؤدي هذا إلى كثير من نقل البيانات غير الضروري، والحل هنا هو إعداد طريقة تنتقل بها الرسائل من عقدة إلى عقدة حتى تصل إلى وجهتها.

تكون الصعوبة في ذلك أنها ستحتاج إلى معرفة بتخطيط الشبكة، حيث من الضروري لإرسال طلب في اتجاه عش بعيد، معرفة أيّ الأعشاش الجارة قادرة على إيصاله بصورة أسرع إلى وجهته، حيث سنهدر كثيرًا من الموارد دون جدوى إذا أرسلناه في الاتجاه الخاطئ.

لا يملك العش المعلومات التي يحتاجها لحساب الطريق لأنه لا يعرف إلا جيرانه المباشَرِين، وعليه يجب نشر معلومات الاتصال لكل الأعشاش بطريقة تسمح لها بالتغير مع مرور الوقت كلما هُجِر عش أو بُني آخر.

نستطيع استخدام الإغراق هنا مرةً أخرى، لكننا سننظر هل تطابق مجموعة الجيران الجديدة لعش ما المجموعة الحالية التي لدينا، أم لا بدلًا من التحقق هل تم استلام الرسالة أم لا.

```
requestType("connections", (nest, {name, neighbors},
                            source) => {
  let connections = nest.state.connections;
  if (JSON.stringify(connections.get(name)) ==
      JSON.stringify(neighbors)) return;
  connections.set(name, neighbors);
  broadcastConnections(nest, name, source);
});

function broadcastConnections(nest, name, exceptFor = null) {
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "connections", {
      name,
      neighbors: nest.state.connections.get(name)
    });
  }
}

everywhere(nest => {
  nest.state.connections = new Map();
  nest.state.connections.set(nest.name, nest.neighbors);
  broadcastConnections(nest, nest.name);
});
```

تستخدِم الموازنة `JSON.stringify` لأنّ `==` لن يُعيد القيمة `true` إلا إذا كان الاثنان نفس القيمة بالضبط سواءً كان على كائنات أو مصفوفات، وليس هذا ما نريده هنا، وبالتالي فقد تكون موازنة سلاسل JSON النصية بدائيةً قليلًا لكنها فعالة لموازنة محتواها.

تبدأ العقد بنشر اتصالاتها مباشرةً، مما يعطي كل عش خريطةً بمخطط الشبكة الحالي، ما لم يوجد عش منها يستحيل الوصول إليه، كما نستطيع باستخدام ذلك المخطط إيجاد الطرق فيه كما رأينا في الفصل السابع، فإذا كان لدينا طريق نحو وجهة رسالة ما، فسنستطيع معرفة أي اتجاه نرسلها فيه.

تبحث الدالة `findRoute` -التي تشابه `findRoute` من الفصل السابع- عن طريقة للوصول إلى عقدة معطاة في الشبكة، لكن بدلًا من إعادة الطريق كله، فهي تعيد الخطوة التالية فقط، وسيقرِّر العش التالي مستخدِمًا المعلومات المتوفرة لديه عن الشبكة أين سيرسل الرسالة تاليًا.

```
function findRoute(from, to, connections) {
  let work = [{at: from, via: null}];
  for (let i = 0; i < work.length; i++) {
    let {at, via} = work[i];
    for (let next of connections.get(at) || []) {
      if (next == to) return via;
      if (!work.some(w => w.at == next)) {
        work.push({at: next, via: via || next});
      }
    }
  }
  return null;
}
```

نستطيع الآن بناء دالة يمكنها إرسال رسائل بعيدة المدى، فإذا كانت مرسَلةً إلى جار مباشر، فستسلَّم مثل العادة؛ أما إذا كان إلى جار بعيد، فستُحزَّم في كائن وترسَل إلى جار قريب من الهدف باستخدام نوع الطلب `"route"` الذي سيجعل ذلك الجار يكرِّر السلوك نفسه.

```
function routeRequest(nest, target, type, content) {
  if (nest.neighbors.includes(target)) {
    return request(nest, target, type, content);
  } else {
    let via = findRoute(nest.name, target,
                        nest.state.connections);
    if (!via) throw new Error(`No route to ${target}`);
    return request(nest, via, "route",
                   {target, type, content});
  }
}
```

```
requestType("route", (nest, {target, type, content}) => {
  return routeRequest(nest, target, type, content);
});
```

نستطيع الآن إرسال الرسالة إلى العش الذي في برج الكنيسة، حيث سنكون قد وفرنا أربع محطات من الطريق في الشبكة.

```
routeRequest(bigOak, "Church Tower", "note",
             "Incoming jackdaws!");
```

لهذا نكون قد بنينا عدة طبقات من الوظائف فوق نظام اتصالات بدائي من أجل تسهيل استخدامه، وهذا نموذج جميل لكيفية عمل شبكات الحواسيب الحقيقية رغم بساطته أو سطحيته، لكن الخاصية التي تميز شبكات الحواسيب أنها لا يُعتمد عليها، فرغم استطاعتنا على بناء تجريدات abstractions فوقها لتساعدنا، إلا أنه لا يمكننا منع فشل الشبكة، لذا تدور برمجة الشبكات حول توقُّع حالات الفشل والتعامل معها.

11.10 دوال async

تنسخ الغربان المعلومات المهمة في الأعشاش من أجل الحفاظ عليها إذا هجم صقر ما على أحدها ودمره، فإذا أرادت استرجاع معلومات ليست موجودة في حوصلة التخزين الخاصة بها، فسيسأل حاسوب العش بقية الأعشاش في الشبكة حتى يجد واحدًا منها لديه تلك المعلومات.

```
requestType("storage", (nest, name) => storage(nest, name));

function findInStorage(nest, name) {
  return storage(nest, name).then(found => {
    if (found != null) return found;
    else return findInRemoteStorage(nest, name);
  });
}

function network(nest) {
  return Array.from(nest.state.connections.keys());
}

function findInRemoteStorage(nest, name) {
  let sources = network(nest).filter(n => n != nest.name);
```

```
    function next() {
      if (sources.length == 0) {
        return Promise.reject(new Error("Not found"));
      } else {
        let source = sources[Math.floor(Math.random() *
                                        sources.length)];
        sources = sources.filter(n => n != source);
        return routeRequest(nest, source, "storage", name)
          .then(value => value != null ? value : next(),
                next);
      }
    }
    return next();
  }
```

ولأن `connections` هي خارطة `Map`، فلن تعمل `Object.keys` عليها، لكن لديها التابع `keys` الذي يعيد مكرِّرًا iterator وليس مصفوفةً، كما يمكن تحويله إلى مصفوفة باستخدام الدالة `Array.from`، وهي شيفرة غير مألوفة حتى مع الوعود، إذ تُسلسَل عدة إجراءات غير متزامنة معًا بطرق غير واضحة، وسنحتاج إلى الدالة التعاودية `next` لوضع نموذج المرور الحلَقي على الأعشاش؛ أما ما تفعله الشيفرة على الحقيقة فهو سلوك خطي، إذ تنتظر تمام الإجراء السابق قبل بدء الذي يليه، وبالتالي ستكون أسهل في التعبير عنها في نموذج برمجة متزامنة.

الجميل في الأمر هو سماح جافاسكربت بكتابة شيفرة وهمية متزامنة pseudo-synchronous لوصف الحوسبة غير المتزامنة، إذ تُعيد الدالة `async` -ضمنيًا- وعدًا وتنتظر `await` في متنها وعودًا أخرى بطريقة تبدو تزامنية.

نُعيد كتابة `findInStorage` كما يلي:

```
  async function findInStorage(nest, name) {
    let local = await storage(nest, name);
    if (local != null) return local;

    let sources = network(nest).filter(n => n != nest.name);
    while (sources.length > 0) {
      let source = sources[Math.floor(Math.random() *
                                      sources.length)];
      sources = sources.filter(n => n != source);
```

```
    try {
      let found = await routeRequest(nest, source, "storage",
                                     name);
      if (found != null) return found;
    } catch (_) {}
  }
  throw new Error("Not found");
}
```

تُحدَّد الدالة `async` بكلمة `async` قبل كلمة `function` المفتاحية، كما يمكن جعل التوابع غير تزامنية بكتابة `async` قبل أسمائها. وإذا استدعيت دالةً مثل تلك؛ فستُعيد وعدًا، وكذلك الحال بالنسبة للتوابع، ويُحل الوعد عندما تُعيد الدالة شيئًا؛ أما إذا رفعت الدالة استثناءً فسيُرفض الوعد.

```
findInStorage(bigOak, "events on 2017-12-21")
.then(console.log);
```

يمكن وضع كلمة `await` المفتاحية أمام تعبير ما داخل دالة `async` لجعله ينتظر حل وعد ما، ولا يتابع تنفيذ الدالة إلا بعد حل ذلك الوعد.

لم تَعُد مثل تلك الدوال تعمل من البداية للنهاية على مرة واحدة، شأنها شأن أيّ دالة جافاسكربت عادية، إذ يمكن تجميدها frozen في أي نقطة توجد فيها `await`، وبعدها ستستعيد العمل في وقت لاحق؛ أما بالنسبة للشيفرات غير المتزامنة المهمة، فتلك الصيغة أسهل من استخدام الوعود مباشرةً، حتى لو احتجنا إلى فعل شيء لا يناسب النموذج المتزامن مثل تنفيذ عدة إجراءات في الوقت نفسه، إذ من السهل دمج `await` مع الاستخدام المباشر للوعود.

11.11 المولدات Generators

إن قدرة الدالة على التوقف ثم المتابعة مرةً أخرى ليست حكرًا على دوال `async` وحدها، حيث تحتوي جافاسكربت على خاصية اسمها دوال المولِّد generator functions التي تشبهها لكن مع استثناء الوعود، فحين نعرِّف دالةً باستخدام `function*` -أي أننا سنضع محرف النجمة بعد الكلمة `function` لتصير مولِّدًا، ويُعيد مكررًا عند استدعائه كما رأينا في الفصل السادس.

```
function* powers(n) {
  for (let current = n;; current *= n) {
    yield current;
  }
}
```

```
for (let power of powers(3)) {
  if (power > 50) break;
  console.log(power);
}
// → 3
// → 9
// → 27
```

تُجمَّد الدالة في بدايتها حين نستدعي powers، وتعمل في كل مرة تستدعي next على المكرِّر حتى تقابل تعبير yield الذي يوقفها ويجعل القيمة المحصَّلة هي القيمة التي ينتجها المكرِّر تاليًا، وحين تُعيد الدالة -علمًا أنّ دالة مثالنا لا تعيد- فسيكون المكرِّر قد انتهى.

تُعَدّ كتابة المكرِّرات أسهل كثيرًا من استخدام الدوال المولِّدة، حيث يمكن كتابة مكرِّر الصنف Group من التدريب في الفصل السادس بهذا المولِّد:

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

لم نعد في حاجة إلى إنشاء كائن ليحفظ حالة المكرِّر، إذ تحفظ المولِّدات حالتها المحلية تلقائيًا في كل مرة تُحصِّل فيها، وقد لا تقع تعبيرات مثل yield تلك إلا مباشرةً في دالة المولِّد نفسها وليس في دالة داخلية معرَّفة داخلها، والحالة التي يحفظها المولِّد عند التحصيل هي بيئته المحلية فقط والموضع الذي حَصَّل فيه؛ أما دالة async فهي نوع خاص من المولِّدات، فهي تُنتِج وعدًا عند استدعائها، كما يُحَل عند إعادتها -أي انتهائها-، في حين يُرفَض إذا رفعت استثناءً، ونحصل على ناتج الوعد سواء كان قيمة أو استثناءً مرفوعًا باستعمال التعبير awaits مع الوعد الذي تعيده الدالة.

11.12 حلقة الحدث التكرارية

تُنفَّذ البرامج غير المتزامنة جزءًا جزءًا، وقد يبدأ كل جزء ببعض الإجراءات ويجدول شيفرات لتنفيذها عند انتهاء الإجراء أو فشله؛ أما بين تلك الأجزاء فسيكون البرنامج ساكنًا منتظِرًا الإجراء التالي، وبالتالي لا تُستدعى ردود النداء من قِبَل الشيفرة التي جدولَتها، فإذا استدعينا setTimeout من داخل دالة، فستكون تلك الدالة قد أعادت بالوقت الذي تُستدعى فيه دالة رد النداء، وحين يُعيد رد النداء، فلن يعود التحكم إلى الدالة التي جدولته.

يحدث السلوك غير المتزامن في مكدس استدعاء الدوال الفارغ الخاص به، وهذا أحد الأسباب التي تصعب معها إدارة الاستثناءات في الشيفرات غير المتزامنة في غياب الوعود، إذ لن تكون معالجات catch الخاصة بنا في المكدس حين ترفع استثناءً بما أنّ كل رد نداء يبدأ بمكدس شبه فارغ.

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (_) {
  // لن يعمل هذا
  console.log("Caught!");
}
```

مهما كانت أوقات وقوع الأحداث -مثل طلبات timeouts أو incoming- متقاربة، فلن تشغّل بيئة جافاسكربت إلا برنامجًا واحدًا في كل مرة.

يمكن النظر إلى هذا السلوك على أنه يشغِّل حلقةً تكراريةً كبيرةً على برنامجنا يُطلق عليها حلقة الحدث event loop، فإذا لم يكن ثمة شيء لفعله فستتوقف الحلقة؛ أما إذا جاءت الأحداث، فستضاف إلى طابور queue وتنفَّذ شيفراتها واحدةً تلو الأخرى، ولأنه لا يمكن تنفيذ شيئين في الوقت نفسه، فستؤخر الشيفرة البطيئة التنفيذ معالجة الأحداث الأخرى.

يضبط المثال أدناه زمن إنهاء timeout، لكن يتلكأ بعد ذلك إلى ما بعد النقطة الزمنية المحدَّدة له مسببًا تأخر زمن الإنهاء.

```
let start = Date.now();
setTimeout(() => {
  console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

تُحل الوعود دائمًا أو تُرفض على أساس حدث جديد، وحتى لو حُل الوعد، فسيتسبب انتظاره في تشغيل رد النداء الخاص بك بعد انتهاء السكربت الحالية بدلًا من تشغيله مباشرةً.

```
Promise.resolve("Done").then(console.log);
console.log("Me first!");
// → Me first!
// → Done
```

سنرى في الفصول اللاحقة عدة أنواع من الأحداث التي تعمل على حلقة الحدث.

11.13 زلات البرامج غير المتزامنة

لا توجد تغيرات حادثة في الحالة إذا كان البرنامج يعمل بتزامن على مرة واحدة عدا تلك التي يصنعها البرنامج نفسه؛ أما بالنسبة للبرامج غير المتزامنة فالأمر مختلف، إذ قد تحتوي على فجوات gaps في تنفيذها يمكن لشيفرات أخرى العمل خلالها.

لننظر إلى المثال التالي، إذ يهوى أحد الغربان عدّ الفراخ التي تفقس في القرية كل عام، حيث تخزِّن الأعشاش هذا العدد في حواصل التخزين الخاصة بها، وتحاول الشيفرة التالية حساب العدد من جميع الأعشاش لأحد الأعوام.

```
function anyStorage(nest, source, name) {
  if (source == nest.name) return storage(nest, name);
  else return routeRequest(nest, source, "storage", name);
}

async function chicks(nest, year) {
  let list = "";
  await Promise.all(network(nest).map(async name => {
    list += `${name}: ${
      await anyStorage(nest, name, `chicks in ${year}`)
    }\n`;
  }));
  return list;
}
```

يُظهر الجزء `name => async` أنّ الدوال السهمية يمكن أن تكون غير متزامنة إذا وضعنا كلمة `async` أمامها. لن يبدو أيّ شيء غريبًا على الشيفرة لأول وهلة، فهي توجه الدوال السهمية غير المتزامنة `async` خلال مجموعة من الأعشاش لتنشئ مصفوفةً من الوعود، ثم ستستخدِم `Promise.all` لتنتظر حل كل تلك الوعود قبل إعادتها القائمة التي جهزتها، غير أنّ هذا السلوك معيب، إذ يُعيد سطرًا واحدًا يحتوي دائمًا على العش الأبطأ -الذي يصل رده متأخرًا- في الإجابة.

```
chicks(bigOak, 2017).then(console.log);
```

تكمن المشكلة في العامِل += الذي يأخذ قيمة list الحالية وقت بدء تنفيذ التعليمة، ثم يضبط رابطة list بعد انتهاء await لتكون تلك القيمة إضافةً إلى السلسلة النصية المضافة؛ أما في الوقت الذي يكون بين بدء تنفيذ التعليمة والوقت الذي تنتهي فيه، سيكون لدينا فجوة غير متزامنة.

يعمل التعبير map قبل أي شيء أضيف إلى القائمة، لذا يبدأ كل عامل من العوامل += بسلسلة فارغة، وينتهي حين تنتهي عملية استعادة التخزين بضبط قائمة list من سطر واحد، حيث تكون نتيجة إضافة السطر إلى السلسلة الفارغة.

كان يمكن تجنب ذلك بسهولة بإعادة الأسطر من الوعود الموجَّهة واستدعاء join على نتيجة Promise.all بدلًا من بناء القائمة بتغيير رابطة ما، كما سيكون حساب قيمة جديدة أقل عرضةً للخطأ من تغيير القيم الموجودة سلفًا.

```
async function chicks(nest, year) {
  let lines = network(nest).map(async name => {
    return name + ": " +
      await anyStorage(nest, name, `chicks in ${year}`);
  });
  return (await Promise.all(lines)).join("\n");
}
```

تحدث أخطاء مثل هذه بسهولة خاصةً عند استخدام await، كما يجب أن نكون على علم بأماكن حدوث الفجوات في شيفراتنا، وإحدى مزايا اللاتزامن الصريح في جافاسكربت، هي أنّ العثور على تلك الفجوات سهل نسبيًا، إذ سيكون إما باستخدام ردود النداء أو الوعود أو await.

11.14 خاتمة

تمكننا البرمجة غير التزامنية من التعبير عن الانتظار للإجراءات actions التي تعمل لوقت طويل دون تجميد البرنامج أثناء تلك الإجراءات، وتنفّذ بيئات جافاسكربت ذلك النمط من البرمجة باستخدام ردود النداء، وهي الدوال التي تُستدعى عند تمام الإجراءات.

تُجدوِل حلقة الحدث ردود النداء لتُستدعى في الوقت المناسب واحدًا تلو الآخر كي لا يتداخل تنفيذها، كما صارت البرمجة غير التزامنية سهلةً بفضل الوعود التي هي كائنات تمثّل إجراءات قد تكتمل في المستقبل، ودوال async التي تسمح لنا بكتابة برنامج غير متزامن كما لو كان تزامنيًا.

11.15 تدريبات

11.15.1 تتبع المشرط

تملك غربان القرية مشرطًا قديمًا تستخدِمه أحيانًا في المهام الخاصة مثل التحزيم أو قطع الأسلاك، كما تريد الحفاظ عليه لئلا يضيع، لذلك، فكلما نُقل المشرط إلى عش جديد أُضيف مُدخل جديد في ذاكرة العش القديم والجديد معًا تحت اسم مشرط `"scalpel"`، حيث تكون قيمته هي موقعه الجديد، ويعني هذا أنّ إيجاد المشرط مسألة اتباع أثر مدخلات الذاكرة حتى تجد عشًا يشير إلى نفسه.

اكتب دالة `async` اسمها `locateScalpel` تفعل ذلك، بحيث تبدأ من العش الذي تشغّله، كما يمكنك استخدام الدالة `anyStorage` التي عرَّفناها من قبل للوصول إلى الذاكرة في أعشاش عشوائية، وافترض أن المشرط قد دار لمدة تكفي لكي يحتوي كل عش على مُدخل `"scalpel"` في ذاكرة بياناته، ثم اكتب الدالة نفسها مرةً أخرى دون استخدام `async` و`await`.

هل تظهر طلبات الفشل مثل رفض للوعود المعادة في كلا النسختين من الدالة؟ وكيف؟

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
async function locateScalpel(nest) {
  // شيفرتك هنا
}

function locateScalpel2(nest) {
  // شيفرتك هنا
}

locateScalpel(bigOak).then(console.log);
// → Butcher Shop
```

إرشادات الحل

يمكن تنفيذ هذا التدريب بحلقة واحدة تبحث في الأعشاش وتنتقل من العش لغيره حين تجد قيمةً لا تطابق اسم العش الحالي، وتعيد اسم العش حين تجد قيمة مطابِقة، كما يمكن استخدام حلقة `while` أو `for` عادية في دالة `async`.

عليك بناء حلقتك باستخدام دالة تعاودية لتنفيذ الشيء نفسه في الدالة المجردة plain function، وأسهل طريقة لذلك هي جعل تلك الدالة تعيد وعدًا باستدعاء `then` على الوعد الذي يجلب قيمة الذاكرة.

سيُعيد المعالج تلك القيمة أو وعدًا مستقبليًا يُنشأ باستدعاء الدالة الحلقية مرةً أخرى بناءً على مطابقة القيمة لاسم العش الحالي.

لا تنسَ بدء الحلقة باستدعاء الدالة التعاودية مرة واحدة من الدالة الأساسية.

تحوِّل `await` الوعود المرفوضة في دالة `async` إلى استثناءات، وحين ترفع دالة `async` استثناءً سيُرفض وعدها، وعليه تكون هذه الطريقة ناجحةً.

إذا استخدمت دالة ليست `async` كما وضحنا سابقًا، فستتسبب الطريقة التي تعمل بها `then` تلقائيًا في وجود الفشل في الوعد المعاد، وإذا فشل الطلب فلن يُستدعى المعالِج الممرَّر إلى `then`، ويُرفض الوعد الذي يعيده للسبب نفسه.

11.15.2 بناء Promise.all

إذا كانت لدينا مصفوفة من الوعود، فإن `Promise.all` تعيد وعدًا ينتظر جميع الوعود الأخرى في المصفوفة حتى انتهائها، حيث سينجح ذلك الوعد ويُحصِّل مصفوفةً من القيم الناتجة، فإذا فشل وعد في المصفوفة، فسيفشل الوعد المعاد من `all` أيضًا، ويكون سبب الفشل هو الوعد الفاشل.

استخدِم شيئًا مثل ذلك بنفسك في صورة دالة عادية اسمها `Promise_all`، وتذكَّر أنّ الوعد لا يمكن أن ينجح أو يفشل بعد نجاحه أو فشله أول مرة، وأنّ الاستدعاءات إلى الدوال التي تحله تُتَجاهل، إذ يُبسِّط ذلك الطريقة التي تعالج بها فشل وعدك.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
function Promise_all(promises) {
  return new Promise((resolve, reject) => {
    // شيفرتك هنا.
  });
}

// شيفرة الاختبار.
Promise_all([]).then(array => {
  console.log("This should be []:", array);
});
function soon(val) {
  return new Promise(resolve => {
    setTimeout(() => resolve(val), Math.random() * 500);
```

```
  });
}
Promise_all([soon(1), soon(2), soon(3)]).then(array => {
  console.log("This should be [1, 2, 3]:", array);
});
Promise_all([soon(1), Promise.reject("X"), soon(3)])
  .then(array => {
    console.log("We should not get here");
  })
  .catch(error => {
    if (error != "X") {
      console.log("Unexpected failure:", error);
    }
  });
```

إرشادات الحل

ستُستدعى `then` من قِبَل الدالة التي تمرَّر إلى باني `Promise` وذلك على كل وعد في المصفوفة المعطاة، وعند نجاح أحد تلك الوعود، فيجب تخزين القيمة الناتجة في الموضع الصحيح في المصفوفة الناتجة، كما يجب التحقق هل كان ذلك هو الوعد الأخير المنتظِر أم لا، وننهي وعدنا الخاص إن كان. ويمكن تنفيذ ذلك التحقق الأخير بعدَّاد يبدأ بطول مصفوفة الدخل، ونطرح منه 1 في كل مرة ينجح فيها أحد الوعود، ونكون قد انتهينا عند وصوله إلى الصفر.

تأكد أن تضع في حسابك موقفًا تكون فيه مصفوفة الدخل فارغة من الوعود، فكيف تُحل الوعود وهي غير موجودة؟ تتطلب معالجة حالات الفشل بعض التأني، لكنها بسيطة في الوقت نفسه، إذ ما عليك سوى تمرير دالة `reject` للوعد المغلِّف إلى كل الوعود الموجودة في المصفوفة على أساس معالِج `catch`، أو على أساس وسيط ثاني إلى `then`، فإذا حدث فشل في أحدها، فسيطلق رفضًا للوعد المغلِّف كله.

بعيد

أكبر موقع توظيف عن بعد في العالم العربي

ابحث عن الوظيفة التي تحقق أهدافك وطموحاتك
المهنية في أكبر موقع توظيف عن بعد

تصفح الوظائف الآن

12. مشروع لغة برمجة

المقيِّم evaluator الذي يحدِّد معاني التعبيرات في لغة البرمجة هو برنامج بذاته.

— هال أبيلسون Hal Abelson وجيرالد سوسْمَن Gerald Sussman، هياكل برامج الحاسوب وتفسيرها.

يُعَدّ بناء لغة برمجة خاصة بك أمرًا ليس بالصعب على عكس ما تتوقع الغالبية الساحقة من الناس، خاصةً إن لم ترفع سقف توقعاتك كثيرًا، بل قد يكون فيه كثير من التعلم النافع أيضًا، والذي نريد الإشارة إليه في هذا الفصل هو أنّ بناء لغة ليس ضربًا من السحر والشعوذة بسبب شعورنا بالإحساس الذي ينتاب المرء في مثل هذه الحالات، فقد جربناه بأنفسنا مع بعض الاختراعات البشرية التي رأينا فيها ذكاءً وصنعةً جعلتنا نظن أنه يستحيل علينا فهم طريقة عملها وبنائها، لكن القراءة والتجربة تفك كثيرًا من هذا السحر.

سنبني في هذا الفصل لغة برمجة سنسميها لغة البيض Egg، حيث ستكون صغيرةً وبسيطةً لكنها قوية بما يكفي لتعبِّر عن أيّ حسابات تفكر فيها، كما ستسمح بالتجريدات abstractions البسيطة المبنية على دوال.

12.1 التحليل

إن أول ما تراه عينك عند النظر إلى لغة برمجة هو بنيتها اللغوية syntax أو صيغتها notation، والمحلل parser هو برنامج يقرأ نصًا ما وينتج هيكل بيانات data structure، بحيث يعكس بنية البرنامج الموجود في ذلك النص، فإن لم يشكِّل النص برنامجًا صالحًا، فيجب أن يخبرنا المحلل بالخطأ الذي سبب ذلك.

ستكون لغتنا ذات بنية لغوية بسيطة وموحدة، حيث سيكون كل شيء في Egg تعبيرًا، وقد يكون التعبير expression اسم رابطة binding أو عددًا أو سلسلةً نصيةً string أو حتى تطبيقًا، كما تُستخدَم التطبيقات لاستدعاءات الدوال وللبنى اللغوية مثل `if` أو `while`.

لن تَدعم السلاسل النصية في Egg أيّ شيء يشبه تهريب الشَرطة المائلة العكسية \، ذلك أنّ السلسلة النصية ببساطة هي سلسلة من محارف داخل اقتباسات مزدوجة، لكن لن تكون هي ذاتها اقتباسات مزدوجة، كما سيكون العدد سلسلةً من الأرقام، ويمكن أن تتكون أسماء الرابطات من أيّ محرف ليس مسافةً فارغةً وليس له معنى خاص في البنية التركيبية للغة.

تُكتَب التطبيقات بالطريقة نفسها المكتوبة بها في جافاسكربت، وذلك بوضع أقواس بعد التعبير، ويمكنها امتلاك أيّ عدد من الوسائط arguments بين هذين القوسين مفصول بين كل منها بفاصلة أجنبية ,.

```
do(define(x, 10),
   if(>(x, 5),
      print("large"),
      print("small")))
```

يعني توحيد لغة Egg أنّ الأشياء التي تُرى على أساس عوامل operators في جافاسكربت -مثل -> هي رابطات عادية في هذه اللغة، وتُطبَّق مثل أيّ دالة أخرى، كما نحتاج إلى بنية do للتعبير عن تنفيذ عدة أشياء في تسلسل واحد، وذلك لعدم وجود مبدأ الكتل blocks في البنية التركيبية للغة.

يتكون هيكل البيانات الذي سيستخدمه المحلل لوصف برنامج ما من كائنات تعابير، لكل منها خاصية type توضِّح نوع التعبير وخصائص أخرى تصف محتواه.

تمثِّل التعابير التي من النوع "value" سلاسل نصيةً مجردةً أو أعدادًا، وتحتوي خاصية value لها على السلسلة النصية أو قيمة العدد الذي تمثله؛ أما التعابير التي من نوع "word" فتُستخدَم للمعرِّفات -أي الأسماء-، فمثل تلك الكائنات لها خاصية name تحمل اسم المعرِّف على أساس سلسلة نصية، وأخيرًا تمثِّل تعابير "apply" التطبيقات، إذ تملك خاصية operator التي تشير مرجعيًا إلى تعبير يُطَبَّق، بالإضافة إلى خاصية args التي تحمل مصفوفةً من تعابير الوسائط argument expressions.

سيُمثَّل جزء >(x, 5) من البرنامج السابق كما يلي:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

تُسمى مثل هذه الهياكل من البيانات باسم شجرة البُنى syntax tree، فإذا تخيلنا الكائنات نقاطًا والروابط التي بينها خطوطًا بين هذه النقاط، فسيكون لدينا شكلًا يشبه الشجرة، فكما أنّ الشجرة بها أغصان تنقسم ويحمل كل منها أغصانًا أخرى؛ فبدوره يشبه احتواء التعابير على تعابير أخرى تحتوي بدورها على تعابير جديدة، وهكذا.

هذا على عكس المحلل الذي كتبناه لصيغة ملف الإعدادات في الفصل التاسع التعابير النمطية، والذي كانت بنيته بسيطةً نوعًا ما، إذ يُقسِّم الدخل إلى أسطر ويعالج هذه الأسطر واحدًا واحدًا، حيث فلم يكن ممكنًا للسطر الواحد إلا بضع صور بسيطة يمكنه أن يكون عليها؛ أما هنا فيجب إيجاد طريقة أخرى، فالتعابير ليست مقسمةً إلى أسطر، كما تملك بنيةً تعاوديةً recursive، وتحتوي تعابير التطبيقات على تعابير أخرى. يمكن حل هذه المشكلة بسهولة من خلال كتابة دالة محلل تعاودية على النحو الذي يعكس الطبيعة التعاودية للغة.

نعرِّف الدالة `parseExpression` التي تأخذ سلسلةً نصيةً على أساس دخل لها، وتُعيد كائنًا يحتوي هيكل البيانات للتعبير في بداية السلسلة النصية مع الجزء المتبقي من السلسلة النصية بعد تحليل هذا التعبير، ويمكن استدعاء هذه الدالة مرةً أخرى حين نحلل التعابير الفرعية كما في حالة وسيط التطبيق مثلًا، وذلك لنحصل على التعبير الوسيط بالإضافة إلى النص المتبقي، وقد يحتوي هذا النص بدوره على وسائط أخرى، أو قد يكون هو قوس الإغلاق في نهاية قائمة الوسائط.

يكون أول جزء في المحلل كالتالي:

```
function parseExpression(program) {
  program = skipSpace(program);
  let match, expr;
  if (match = /^"([^"]*)"/.exec(program)) {
    expr = {type: "value", value: match[1]};
  } else if (match = /^\d+\b/.exec(program)) {
    expr = {type: "value", value: Number(match[0])};
```

```
  } else if (match = /^[^\s(),#"]+/.exec(program)) {
    expr = {type: "word", name: match[0]};
  } else {
    throw new SyntaxError("Unexpected syntax: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

يجب قص المسافات الفارغة من بداية السلسلة النصية للبرنامج بما أنّ لغة Egg التي نكتبها تشبه جافاسكربت في كونها تسمح لأيّ عدد من المسافات الفارغة بين عناصرها، وهنا يأتي دور دالة `skipSpace`.

تَستخدم `parseExpression` بعد تخطي أي مسافة بادئة ثلاثة تعبيرات نمطية لتحديد العناصر الصغرى الثلاثة التي تدعمها Egg، وهي السلاسل والأعداد والكلمات، كما يبني المحلل نوعًا مختلفًا من هياكل البيانات وفقًا للتي تطابق منها، فإذا كان الإدخال لا يتطابق مع أحد هذه النماذج الثلاثة، فلا يكون تعبيرًا صالحًا ويرفع المحلل خطأً.

نحن نستخدم `SyntaxError` بدلًا من `Error` على أساس باني استثناءات exception، وهو نوع قياسي آخر للأخطاء لأنه أكثر تحديدًا، وهو أيضًا نوع الخطأ الذي يرمى عند محاولة تشغيل برنامج JavaScript غير صالح، ثم نقص الجزء المطابق من سلسلة البرنامج النصية ونمرره مع كائن التعبير إلى `parseApply` الذي ينظر هل التعبير تطبيق أم لا، فإذا كان تطبيقًا، فسيحلِّل قائمةً من الوسائط محصورةً بين قوسين.

```
function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(") {
    return {expr: expr, rest: program};
  }

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
```

```
      let arg = parseExpression(program);
      expr.args.push(arg.expr);
      program = skipSpace(arg.rest);
      if (program[0] == ",") {
        program = skipSpace(program.slice(1));
      } else if (program[0] != ")") {
        throw new SyntaxError("Expected ',' or ')'");
      }
    }
    return parseApply(expr, program.slice(1));
  }
```

إذا كان المحرف التالي في البرنامج ليس قوسًا بادئًا) فلن يكون هذا تطبيقًا، وتُعيد parseApply التعبير المعطى لها، وإلا فستتخطى القوس البادئ وتنشئ كائن شجرة بُنى syntax tree object لتعبير التطبيق ذاك، ثم تستدعي parseExpression تعاوديًا لتحلِّل كل وسيط حتى تجد القوس الغالق، كما يكون التعاود هنا غير مباشر من خلال استدعاء parseApply لدالة parseExpression والعكس، ولأن تعبير التطبيق يمكن تطبيقه كما في multiplier(2)(1)، فيجب أن تستدعي parseApply نفسها مرةً أخرى لتنظر فيما إذا كان يوجد زوج آخر من الأقواس أم لا، وذلك بعد تحليل تطبيق ما.

هذا كل ما نحتاج إليه لتحليل لغة Egg، إذ نغلفها في دالة parse لتتحقق أنها وصلت إلى نهاية سلسلة الدخل بعد تحليل التعبير -فبرنامج بلغة Egg هو تعبير وحيد-، ثم تعطينا هيكل البيانات للبرنامج.

```
function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(rest).length > 0) {
    throw new SyntaxError("Unexpected text after program");
  }
  return expr;
}

console.log(parse("+(a, 10)"));
// → {type: "apply",
//    operator: {type: "word", name: "+"},
//    args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]}
```

وهكذا تعمل اللغة بنجاح، رغم أنها لا تعطينا معلومات مفيدة حين تفشل أو تتعطل، كما لا تخزِّن السطر والعمود الذي يبدأ عنده كل تعبير -وهو الأمر الذي قد يكون مفيدًا عند الإبلاغ عن الأخطاء فيما بعد-، لكن لا يهم، فما أنجزناه إلى الآن كافي.

12.2 التقييم

لا شك أننا نريد شجرة بنى لبرنامج ما من أجل تشغيلها، وهذه هي وظيفة المقيِّم evaluator، إذ تعطيه شجرة بنى وكائن نطاق يربط الأسماء بالقيم، وسيقيِّم التعبير الذي تمثله الشجرة، ثم يُعيد القيمة التي يخرجها.

```
const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expr.name}`);
    }
  } else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expr.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

يحتوي المقيِّم على شيفرة لكل نوع من أنواع التعابير، ويُخرج لنا تعبير القيمة مصنفة النوع literal value expression قيمته، كما في حالة التعبير 100 الذي يقيِّم إلى العدد 100 فقط؛ أما في حالة الرابطة، فيجب التحقق مما إذا كانت معرَّفةً على الحقيقة في النطاق أم لا، وإن كانت فسنجلب قيمتها.

تُعَدّ التطبيقات أكثر تفصيلًا من ذلك، فإذا كانت صيغةً خاصةً مثل `if`، فلا نقيِّم أيّ شيء ونمرِّر التعابير الوسيطة مع النطاق إلى الدالة التي تعالج هذه الصيغة؛ أما إن كانت استدعاءً عاديًا، فسنقيِّم العامِل ونتحقق أنه دالة، ونستدعيها مع الوسائط المقيَّمة.

نستخدِم قيمًا لدوال جافاسكربت عادية لنمثِّل قيم الدوال في Egg، كما سنعود إلى هذا بعد قليل حين تُعرَّف الصيغة الخاصة المسماة `fun`، ويمثِّل الهيكل التعاودي لـ `evaluate` هيكلًا مماثلًا للمحلِّل، وكلاهما يعكس هيكل اللغة نفسه، ومن الممكن مكاملة المحلِّل والمقيِّم أثناء التحليل أيضًا، لكن فصلهما عن بعضهما البعض بهذه الطريقة يجعل البرنامج أكثر نقاءً.

هذا جل ما نحتاج إليه لتفسير لغة Egg ببساطة، لكن من ناحية أخرى، لا تستطيع فعل الكثير بهذه اللغة دون تعريف بعض الصيغ الخاصة الأخرى وإضافة بعض القيم المفيدة إلى البيئة.

12.3 الصيغ الخاصة

يُستخدَم كائن الصيغ الخاصة `specialForms` لتعريف البُنى الخاصة في لغة Egg، حيث يربط الكلمات بالدوال التي تقيِّم مثل تلك الصيغ، ولنضف `if` بما أنه فارغ الآن:

```
specialForms.if = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Wrong number of args to if");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

تتوقع بنية `if` ثلاثة وسائط بالضبط، وستقيِّم الأول منها، فإذا لم تكن النتيجة هي القيمة `false` فستقيِّم الثاني، وإلا فالثالث هو الذي يُقيَّم، وتُعَد صيغة `if` هذه أقرب إلى عامل `?:` الثلاثي في جافاسكربت منها إلى عامل `if` في جافاسكربت أيضًا، فهو تعبير وليس تعليمة، ويُنتِج قيمةً تكون نتيجة الوسيط الثاني أو الثالث.

تختلف كذلك لغة Egg عن جافاسكربت في كيفية معالجة القيمة الشرطية إلى `if`، فهي لا تعامل الصفر والسلسلة النصية الفارغة على أنها خطأ false، بل القيمة `false` حصرًا وفقط.

السبب الذي يجعلنا في حاجة إلى تمثيل `if` على أساس صيغة خاصة بدلًا من دالة عادية، هو أن جميع وسائط الدوال تُقيَّم قبل استدعاء الدالة، بينما يجب ألا تقيِّم `if` إلا وسيطها الثاني أو الثالث بناءً على قيمة الوسيط الأول.

صيغة `while` شبيهة بهذا، فبما أن `undefined` غير موجودة في لغة Egg، فسنُعيد `false` لعدم وجود نتيجة أفضل وأكثر فائدة، كما في المثال التالي:

```
specialForms.while = (args, scope) => {
  if (args.length != 2) {
    throw new SyntaxError("Wrong number of args to while");
  }
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }
  return false;
};
```

كذلك لدينا وحدة بنيوية أساسية أخرى هي `do` التي تنفِّذ كل وسائطها من الأعلى إلى الأسفل، وتكون قيمتها هي القيمة التي ينتجها الوسيط الأخير.

```
specialForms.do = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
  return value;
};
```

ننشئ صيغةً نسميها `define` كي نستطيع إنشاء رابطات ونعطيها قيمًا جديدةً، حيث تتوقع هذه الصيغة للوسيط الأول لها أن يكون كلمةً وتعبيرًا ينتج القيمة التي سنسندها إلى تلك الكلمة لتكون الوسيط الثاني، وبما أنّ `define` ما هي إلا تعبير، فيجب أن تُعيد قيمةً ما، وسنجعلها تُعيد القيمة التي أُسندت إليها كما في حالة عامل `=` في جافاسكربت.

```
specialForms.define = (args, scope) => {
  if (args.length != 2 || args[0].type != "word") {
    throw new SyntaxError("Incorrect use of define");
  }
```

```
    let value = evaluate(args[1], scope);
    scope[args[0].name] = value;
    return value;
  };
```

12.4 البيئة

يكون النطاق الذي تقبَله evaluate كائنًا له خصائص تتوافق أسماؤها مع أسماء الرابطات، كما تتوافق قيمها مع القيم التي ترتبط بهذه الرابطات، وسنعرِّف كائنًا ليمثل النطاق العام.

يجب أن تكون لنا وصول إلى قيم بوليانية كي نستطيع استخدام بنية if التي عرفناها لتونا، وبما أنه لا يوجد إلا قيمتان منطقيتان فقط، فلا نحتاج إلى ترميز خاص لهما، بل نربطهما بالقيمتين true وfalse ثم نستخدمهما.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

نستطيع الآن تقييم تعبير بسيط يرفض قيمةً بوليانيةً.

```
let prog = parse(`if(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

سنضيف بعض قيم الدوال إلى النطاق لتوفير العوامل الحسابية الأساسية وكذلك عوامل الموازنة، كما سنستخدم Function -بهدف تبسيط الشيفرة والحفاظ على قصرها- لتوليف مجموعة من دوال العوامل في حلقة تكرارية بدلًا من تعريف كل واحد منها على حدة.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

إذا كانت لدينا طريقة لإخراج القيم، فسيكون ذلك مفيدًا لنا بلا شك، لذا سنغلِّف console.log في دالة ونسميها print.

```
topScope.print = value => {
  console.log(value);
```

```
  return value;
};
```

يعطينا هذا الأدوات الأساسية اللازمة لكتابة برامج بسيطة، حيث توفر الدالة أدناه طريقةً سهلةً لتحليل برنامج ما وتشغيله في نطاق جديد:

```
function run(program) {
  return evaluate(parse(program), Object.create(topScope));
}
```

سنستخدم سلاسل كائن النموذج الأولي لتمثيل النطاقات المتشعِّبة nested scopes كي يتمكن البرنامج من إضافة روابط إلى نطاقه المحلي دون تغيير النطاق الأعلى top-level scope.

```
run(`
do(define(total, 0),
   define(count, 1),
   while(<(count, 11),
         do(define(total, +(total, count)),
            define(count, +(count, 1)))),
   print(total))
`);
// → 55
```

ذلك هو البرنامج الذي رأيناه عدة مرات من قبل، والذي يحسب مجموع الأرقام من 1 إلى 10 مكتوبًا بلغة Egg، ومن الواضح أنه ليس أجمل من مثيله في جافاسكربت، لكنا نراه ممتازًا إذا نظرنا إلى حقيقة أنّ اللغة التي كُتب بها ليس فيها إلا 150 سطرًا من الشيفرات فقط.

12.5 الدوال

تُعَدّ لغة البرمجة التي ليس فيها دوال لغةً فقيرةً في إمكاناتها، لكن لحسن الحظ فليس من الصعب إضافة بنية مثل `fun` التي تعامِل وسيطها الأخير على أنه متن الدالة، وستستخدم جميع الوسائط التي قبله على أساس أسماء لمعامِلات الدالة.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
```

```
    let params = args.slice(0, args.length - 1).map(expr => {
      if (expr.type != "word") {
        throw new SyntaxError("Parameter names must be words");
      }
      return expr.name;
    });

    return function() {
      if (arguments.length != params.length) {
        throw new TypeError("Wrong number of arguments");
      }
      let localScope = Object.create(scope);
      for (let i = 0; i < arguments.length; i++) {
        localScope[params[i]] = arguments[i];
      }
      return evaluate(body, localScope);
    };
  };
```

تحصل الدوال في لغة Egg على نطاقاتها المحلية الخاصة بها، إذ تُنشِئ الدالة المنتَجة بواسطة صيغة fun هذا النطاق المحلي، وتضيف الرابطات الوسيطة إليه، ثم تقيِّم متن الدالة في هذا النطاق وتُعيد النتيجة.

```
  run(`
  do(define(plusOne, fun(a, +(a, 1))),
     print(plusOne(10)))
  `);
  // → 11

  run(`
  do(define(pow, fun(base, exp,
       if(==(exp, 0),
          1,
          *(base, pow(base, -(exp, 1)))))),
     print(pow(2, 10)))
  `);
  // → 1024
```

12.6 التصريف

يسمى الذي بنيناه حتى الآن بالمفسِّر interpreter، والذي يتصرف مباشرةً أثناء التقييم على تمثيل البرنامج الذي أنتجه المحلِّل؛ أما التصريف compilation، فهو عملية إضافة خطوة أخرى بين التحليل وتشغيل البرنامج، إذ تحوِّل البرنامج إلى شيء يمكن تقييمه بكفاءة أكبر من خلال إضافة كل ما يمكن إضافته من عمل ومهام مقدمًا، فمن الواضح في اللغات متقَنة التصميم مثلًا استخدام كل رابطة وأيّ رابطة مشار إليها بدون تشغيل البرنامج فعليًا، حيث يُستخدَم هذا لتجنب البحث عن الرابطة باسمها في كل مرة يوصَل إليها، بدلًا من جلبها مباشرةً من موقع محدد سلفًا في الذاكرة.

يتضمن التصريف عادةً تحويل البرنامج إلى لغة الآلة، وهي الصيغة الخام التي يستطيع معالج الحاسوب تنفيذها، لكن هذه ليست الصورة الوحيدة له، فأيّ عملية تحوِّل البرنامج إلى تمثيل آخر مختلف يمكن النظر إليها على أنها تصريف كذلك.

سيكون من الممكن كتابة خطة تقييم بديلة للغة Egg، إذ تبدأ بتحويل البرنامج إلى برنامج جافاسكربت أولًا، ثم تستخدِم `Function` لاستدعاء مصرِّف جافاسكربت له، وبعدها سنحصل على النتيجة، فإذا كان هذا بكفاءة، فسيجعل لغة Egg تعمل بمعدل سريع جدًا مع الحفاظ على بساطة الاستخدام.

12.7 التغليف

لعلك لاحظت حين عرَّفنا `if` و `while` على أنهما ليستا سوى تغليف بسيط نوعًا ما حول نظيرتيهما في جافاسكربت، وبالمثل، فإنّ القيم في Egg ما هي إلا قيم جافاسكربت العادية.

إذا وازنت استخدام Egg المبني على جافاسكربت مع كمية الجهد والتعقيد المطلوب لبناء لغة برمجة من الوظائف المجردة الخام التي يوفرها الحاسوب على أساس عتاد، فإنّ الفارق عظيم، وعلى كل حال يعطيك هذا المثال صورةً للطريقة التي تعمل بها لغة البرمجة.

إذا وضعنا هذا في ميزان الإنجاز، فستكون الاستفادة من العجلة الموجودة أفضل من إعادة اختراعها وتنفيذ كل شيء بأنفسنا، رغم أنّ هذه اللغة التي أنشأناها في هذا الفصل والتي تشبه لعب الأطفال لا تفعل أي شيء لا تفعله جافاسكربت بالكفاءة نفسها إن لم يكن أفضل، إلا أن هناك حالات سيكون من المفيد فيها كتابة برامج بسيطة لإنجاز المهام التي بين أيدينا، ومثل هذه اللغة ليس عليها أن تكون على صورة اللغة التقليدية، فإذا لم تأتي جافاسكربت بتعابير نمطية مثلًا، فستستطيع كتابة محلِّل خاص بك ومقيِّم كذلك من أجل هذه التعابير؛ أو لنقل أنك تبني ديناصورًا آليًا عملاقًا وتريد برمجة سلوكه، فقد لا تكون جافاسكربت حينها هي اللغة المثلى لذلك، وستجد أنك تحتاج إلى لغة تبدو هكذا:

```
behavior walk
  perform when
    destination ahead
  actions
    move left-foot
    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets
```

يُطلَق على مثل هذه اللغة أنها لغة مختصة بمجال بعينه domain-specific، وهي لغة مفصَّلة لتعبِّر عن عناصر مجال معين من المعرفة، ولا تتعداه إلى غيره، كما تكون أكثر وصفًا وإفصاحًا عن اللغة الموجَّهة للأغراض العامة لأنها مصمَّمة لتصف الأشياء المراد وصفها حصرًا في هذا المجال وحسب.

12.8 تدريبات

12.8.1 المصفوفات

اجعل لغة Egg تدعم المصفوفات من خلال إضافة الدوال الثلاثة التالية إلى النطاق العلوي top scope:

- `array(...values)` لبناء مصفوفة تحتوي على قيم وسيطة.
- `length(array)` للحصول على طول مصفوفة ما.
- `element(array, n)` لجلب العنصر رقم n من مصفوفة ما.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// عدِّل هذه التعريفات...

topScope.array = "...";

topScope.length = "...";
```

```
topScope.element = "...";

run(`
do(define(sum, fun(array,
     do(define(i, 0),
        define(sum, 0),
        while(<(i, length(array)),
          do(define(sum, +(sum, element(array, i))),
             define(i, +(i, 1)))),
        sum))),
   print(sum(array(1, 2, 3))))
`);
// → 6
```

إرشادات الحل

إن أسهل طريقة لإضافة الدعم هي تمثيل مصفوفات Egg بمصفوفات جافاسكربت، ويجب أن تكون القيم المضافة إلى النطاق العلوي دوالًا، كما يمكن أن يكون تعريف `array` بسيطًا جدًا إذا استَخدَمت وسيط rest مع الصيغة ثلاثية النقاط triple-dot notation.

12.8.2 التغليف Closure

تسمح لنا الطريقة التي عرَّفنا بها `fun` بإضافة دوال في Egg للإشارة مرجعيًا إلى النطاق المحيط، مما يسمح لمتن الدالة أن تستخدِم قيمًا محليةً كانت مرئيةً في وقت تعريف الدالة، تمامًا مثلما تفعل دوال جافاسكربت، ويوضِّح البرنامج التالي هذا، إذ تُعيد دالة `f` دالةً تضيف وسيطها إلى وسيط `f`، مما يعني أنها تحتاج إلى وصول للنطاق المحلي الموجود داخل `f` كي تستطيع استخدام الرابطة `a`.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
run(`
do(define(f, fun(a, fun(b, +(a, b)))),
   print(f(4)(5)))
`);
// → 9
```

ارجع الآن إلى تعريف صيغة `fun` واشرح الآلية المسببة لعملها.

إرشادات الحل

سنستخدِم آليات جافاسكربت مرةً أخرى للحصول على مزايا مشابهة في Egg، حيث تُمرَّر الصيغ الخاصة إلى النطاق المحلي الذي تقيَّم فيه كي تقيِّم هي صيغها الفرعية في ذلك النطاق، ويكون للدالة التي تعيدها fun وصول إلى وسيط scope المعطى للدالة المغلِّفة، كما تستخدِم ذلك لإنشاء نطاق الدالة المحلي حين تُستدعى.

يعني هذا أن النموذج الأولي للنطاق المحلي سيكون هو النطاق الذي أُنشَئت فيه الدالة مما يمكننا من الوصول إلى الرابطات التي في ذلك النطاق من خلال الدالة، وهذا كله من أجل استخدام التغليف closure رغم أنك في حاجة إلى مزيد من العمل لتُصرِّف ذلك بكفاءة.

12.8.3 التعليقات

أليس من الجميل أن تتمكن من كتابة تعليقات في لغة Egg؟ فلو جعلنا التعليق يبدأ بعلامة # مثلًا كما في حال علامتي // في جافاسكربت -ولا تقلق بشأن المحلِّل، إذ لن نجري أيّ تغييرات جوهرية فيه كي يدعم التعليقات-، فما علينا سوى تغيير skipspace كي تتخطى التعليقات كذلك، كما لو كانت مسافات فارغة، بحيث نتخطى التعليقات في كل مرة نستدعي فيها skipspace.

الشيفرة أدناه تمثل skipspace، عدِّلها لتضيف دعم التعليقات في لغة Egg، مسترشدًا بما سبق.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}

console.log(parse("# hello\nx"));
// → {type: "word", name: "x"}

console.log(parse("a # one\n   # two\n()"));
// → {type: "apply",
//    operator: {type: "word", name: "a"},
//    args: []}
```

إرشادات الحل

تأكد من جعل حلّك قادرًا على معالجة التعليقات المتعددة في صف مع مسافات فارغة محتملة بينها أو بعدها. سيكون التعبير النمطي أسهل طريقة تستخدمها في هذا التمرين للحل، لهذا اكتب شيئًا يطابق "مسافةً فارغةً أو تعليقًا أو صفرًا أو أكثر من مرة". واستخدم التابع `exec` أو`match`، وانظر إلى طول أول عنصر في المصفوفة المعادة -أي التطابق الكامل- لتعرف كم عدد المحارف التي عليك قصها.

12.8.4 إيجاد النطاق

الطريقة الوحيدة حاليًا لإسناد قيمة إلى رابطة هي `define`، حيث تتصرف هذه البنية مثل طريقة لتعريف رابطات جديدة وإعطاء قيمة جديدة للرابطات الحالية.

لكن هذا الإبهام يجعلك تقع في مشكلة تعريف رابطة محلية بالاسم نفسه في كل مرة تحاول إعطاء رابطة غير محلية قيمةً جديدةً، كما نستنكر هذه الطريقة في معالجة النطاقات رغم وجود لغات مصممة على هذا الأساس. أضف صيغة `set` الخاصة التي تشبه `define`، وتعطي قيمةً جديدةً للرابطة لتحدِّث الرابطة في نطاق خارجي إذا لم تكن موجودةً سلفًا في النطاق الداخلي، وإن لم تكن هذه الرابطة معرفةً، فأبلغ بالخطأ `ReferenceError` الذي هو نوع خطأ قياسي. سيعيقك نوعًا ما أسلوب تمثيل النطاقات على أساس كائنات بسيطة من أجل السهولة ها هنا، خاصةً إذا أردت استخدام دالة `Object.getPrototypeOf` مثلًا التي تعيد النموذج الأولي لكائن ما، وتذكَّر أيضًا أنّ النطاقات لا تنحدر من `Object.prototype`، لذا فإن أردت استدعاء `hasOwnProperty` عليها، فعليك استخدام التعبير التالي:

```
Object.prototype.hasOwnProperty.call(scope, name);
specialForms.set = (args, scope) => {
  // ضع شيفرتك هنا.
};

run(`
do(define(x, 4),
   define(setx, fun(val, set(x, val))),
   setx(50),
   print(x))
`);
// → 50
run(`set(quux, true)`);
// → ReferenceError أحد صور
```

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

إرشادات الحل

سيكون عليك التكرار على نطاق واحد في كل مرة باستخدام `Object.getPrototypeOf` للذهاب إلى النطاق الخارجي اللاحق. واستخدِم `hasOwnProperty` لتعرف ما إذا كانت الرابطة الموضَّحة بخاصية `name` في أول وسيط لـ `set` موجودةً في ذلك النطاق أم لا، فإن كانت كذلك فاضبطها على نتيجة تقييم الوسيط الثاني لـ `set` ثم أعد تلك القيمة، وإذا وصلت إلى أقصى نطاق خارجي -بحيث تكون إعادة `Object.getPrototypeOf` هي `null`- ولم تعثر على الرابطة بعد، فستكون هذه الرابطة غير موجودة ويجب رمي خطأ هنا.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

13. جافاسكربت والمتصفحات

يُعَدّ توفير فضاء مشترك للمعلومات نتواصل فيه من خلال مشاركة هذه المعلومات حلمًا بُنيت عليه الشبكة العنكبوتية (الويب)، كما تُعَدّ عمومية هذه الشبكة لازمةً وضروريةً، إذ يستطيع رابط النصوص التشعبي hypertext link الإشارة إلى أي شيء سواءً كان شخصيًا أو محليًا أو عالميًا، مسودةً كان أو نسخةً منقحةً.

— تِم بِرنَرز لي Tim Berners-Lee، الشبكة العنكبوتية العالمية: تاريخ شخصي قصير للغاية.

سنتحدث في الفصول الباقية من هذا الكتاب عن متصفحات الويب، فبدونها لم تكن جافاسكربت لتكون أصلًا، وإذا وُجدت لسبب ما فلم يكن أحد ليلتفت إليها.

بُنيت تقنية الشبكات لتكون غير مركزية من البداية، سواءً على الصعيد الفني أو بالنسبة للطريقة التي تطورت بها، فقد أضافت العديد من الجهات الموفِّرة للمتصفحات وظائف جديدةً لأغراض محددة بعينها أحيانًا، وبدون تفكير أحيانًا أخرى لتعطينا وظائف ومزايا جديدة في صور سيئة، ثم يدور الزمن بهذه وتلك إلى أن يتبناها أحد ما ثم تصير معيارًا قياسيًا في النهاية.

ونرى أنّ هذا قد يكون نعمةً ونقمةً في الوقت نفسه، فمن الجيد ألا يكون لديك تحكم مركزي في نظام مثل الإنترنت ويكون تطوره على يد جهات مختلفة تتعاون أحيانًا فيما بينها بصورة طفيفة، وتكاد تناوش بعضها أحيانًا أخرى؛. لكن من الناحية الأخرى، تعني الطريقة العفوية التي تطور بها الإنترنت أنّ النظام الناتج لن يكون هو الآلية التي يتوقعها البعض من حيث الاتساق النظامي الداخلي، فبعض أجزاء هذا النظام مشوشة إلى حد الحيرة.

13.1 الشبكات والإنترنت

وُجِدت شبكات الحواسيب منذ خمسينات القرن الماضي، ذلك أنك إذا وصلت حاسوبين أو أكثر بسلك لنقل البيانات فيما بينها، فستستطيع فعل الأعاجيب بهذه الشبكة الصغيرة، وعليه فهذه الأعاجيب تزداد عجبًا حين نوصل جميع الحواسيب في العالم ببعضها البعض.

وقد بدأت التقنية التي تطبق هذه الرؤية في التطوير في الثمانينيات، ثم حصلنا على الشبكة التي تسمى بالإنترنت، وقد كانت كما حلمنا بها بالضبط.

يستخدِم الحاسوب هذه الشبكة ليرسل بِتَّات من البيانات إلى حاسوب آخر، ويجب على كلا الحاسوبين معرفة ماذا يفترض لهذه البِتَّات أن تكون وماذا تمثل من أجل تحقيق تواصل فعال، ويتوقف معنى أي تسلسل من البتات على نوع الشيء المراد التعبير عنه وعلى آلية الترميز encoding mechanism المستخدَمة.

يصف بروتوكول الشبكة network protocol أسلوبًا من التواصل عبر أي شبكة، فهناك بروتوكولات لإرسال البريد الإلكتروني وجلبه ومشاركة الملفات، وحتى التحكم في الحواسيب التي قد تكون مصابةً ببرمجيات خبيثة، فيُستخدَم بروتوكول نقل النصوص الفائقة HTTP مثلًا -وهو اختصار لـ Hypertext Transfer Protocol- لجلب الموارد المسمّاة وكتل المعلومات مثل صفحات الويب أو الصور، كما يشترط على الجزء الذي سينشئ الطلب البدء بسطر يشبه السطر التالي مسميًا المورد وإصدار البروتوكول الذي يريد استخدامه:

```
GET /index.html HTTP/1.1
```

هناك قواعد كثيرة تحكم الطريقة التي يمكن للطالب requester فيها إدخال بيانات أو معلومات في الطلب، والطريقة التي يحزِّم الطرف الآخر بها الموارد المطلوبة، وهو مستقبِل الطلب الذي يعيد هذه الموارد إلى طالبها، كما سننظر بشيء من التفصيل في بروتوكول HTTP في الفصل الثامن عشر.

تُبنى أغلب البروتوكولات على بروتوكولات أخرى، حيث يتعامل بروتوكول HTTP مع الشبكة كما لو كانت أداة يضع فيها البِتّات ويتوقع منها الوصول إلى الوجهة الصحيحة بالترتيب المتوقع لها، لكن الواقع المشاهَد يقول عكس ذلك كما رأينا في الفصل الحادي عشر. ويعالِج بروتوكول التحكم في النقل TCP -اختصارًا لـ Transmission Control Protocol- هذه المشكلة بما أن كل الحواسيب المتصلة بالإنترنت تفهمه، وقد بُنيت أغلب عمليات التواصل في الإنترنت عليه أصلًا.

فكرة عمل بروتوكول TCP هي أنّ الحاسوب يجب أن يكون في وضع انتظار أو استماع للحواسيب الأخرى حتى تكلمه أو تراسله، كما يجب أن يكون لكل مستمِع رقمًا يسمى منفَذًا port ويرتبط به من أجل أن تستطيع استماع عمليات تواصل مختلفة في الوقت نفسه على آلة واحدة، كما تحدد أغلب البروتوكولات المنفَذ الذي يجب استخدامه افتراضيًا، فحين نريد إرسال بريد إلكتروني باستخدام بروتوكول SMTP مثلًا، فيجب أن تستمع الآلة التي نرسله من خلالها إلى المنفَذ 25، ثم سينشئ حاسوب آخر حينئذ اتصالًا من خلال التوصيل بالآلة

الهدف مستخدمًا رقم المنفَذ الصحيح، فإذا استطاع الوصول إلى الآلة الهدف وكانت تستمع لهذا المنفَذ، فسنقول أنّ الاتصال تحقق بنجاح، كما يُسمى الحاسوب المستمع هنا بالخادم server والحاسوب المتصَل به بالعميل client.

يعمل مثل ذلك الاتصال على أساس خط أنابيب ثنائي الاتجاه يمكن للبِتّات أن تتدفق فيه من كلا الاتجاهين، مما يعني استطاعة كلا الحاسوبين إرسال البيانات واستلامها عبره، وبمجرد أن تُنقل البِتّات بنجاح، فسيمكن للآلة التي على الطرف الآخر قراءتها مرةً أخرى، وهذا النموذج مريح جدًا، بل يمكن القول أنّ TCP هو النموذج المجرَّد للشبكة.

13.2 الشبكة العنكبوتية العالمية The Web

تُعَدّ الشبكة العنكبوتية العالمية -أو الويب- مجموعةً من البروتوكولات والصيغ التي تسمح لنا بزيارة صفحات الويب داخل متصفح ما، وتشير كلمة الشبكة العنكبوتية فيها إلى حقيقة إشارة هذه الصفحات إلى بعضها بعضًا وارتباطها ببعضها، وبالتالي تكوّن شبكةً عملاقةً يستطيع المستخدِم التنقل خلالها بحُرية، كما يجدر بنا الإشارة هنا إلى أنّ هذه الشبكة ليست هي كل الإنترنت، فالأخير أوسع وأشمل.

لكي تكون جزءًا من هذه الشبكة، لا تحتاج إلا إلى توصيل آلة بالإنترنت وتجعلها تستمع إلى المنفَذ 80 ببروتوكول HTTP كي تستطيع الحواسيب الأخرى طلب مستندات وملفات منها، ويُسمى كل مستند على الشبكة بمحدِّد موقع الموارد الموحَّد Uniform Resource Locator -أو URL اختصارًا-، حيث يكون أشبه بما يلي:

```
 http://eloquentJavaScript.net/13_browser.html
|      |                      |               |
protocol       server               path
```

يخبرنا أول جزء أنّ هذا الرابط يستخدم بروتوكول HTTP على خلاف بروتوكول HTTP المشفَّر مثلًا التي سيكون هكذا: `https://`، ثم يعرِّف الجزءُ الذي يليه الخادمَ الذي نطلب منه المستند، ولدينا في النهاية مسار من سلسلة نصية يعرِّف المستند المحدَّد أو المورد الذي نريد.

تحصل الآلات المتصلة بالإنترنت على عنوان IP يكون رقمًا يمكن استخدامه لإرسال رسائل إلى هذه الآلة، حيث يبدو هكذا: `149.210.142.219` أو `2001:4860:4860::8888`، غير أن هذه الأرقام المتتالية يصعب حفظها وكتابتها، لهذا تستطيع تسجيل اسم نطاق لأحد تلك العناوين أو بعضها، وقد سجلنا eloquentJavaScript.net مثلًا لنشير إلى عنوان IP لآلة نتحكم فيها ونستطيع استخدام اسم النطاق حينئذ لإرسال صفحات الويب لمن يطلبها.

إذا كتبتَ محدِّد المواقع ذاك في شريط عنوان المتصفح، فسيحاول المتصفح جلب المستند الذي في ذلك الرابط وعرضه، فيبدأ بالنظر في العنوان الذي يشير إليه eloquentJavaScript.net، ثم يستخدِم

بروتوكول HTTP ليحقق اتصالًا بالخادم الذي في ذلك العنوان ويطلب المورِد `/13_browser.html`، وإذا تم ذلك كله فسيرسل الخادم حينئذ مستندًا يعرضه لك المتصفح على الشاشة.

13.3 HTML

تُهيأ صفحات الويب التي تراها أمامك على الشاشة بتنسيق يتكون من تراكيب محددة، واللغة التي نكتب بها تلك التراكيب اسمها HTML أو Hypertext Markup Language وتعني لغة ترميز النصوص الفائقة، ويحتوي المستند العادي المكتوب بهذه اللغة على نصوص ووسوم tags، وتحدد التركيب البنائي لتلك النصوص كما تصف الأنواع المختلفة لها سواءً كانت روابط أو فقرات أو عناوين، وتبدو هذه اللغة هكذا:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentJavaScript.net">here</a>.</p>
  </body>
</html>
```

توفِّر الوسوم معلومات عن بنية المستند وتحاط بقوسين زاويين هما رمزا "أصغر من" و"أكبر من" في الرياضيات كما ترى في المثال أعلاه؛ أما بقية النص فيكون نصًا عاديًا، كما يبدأ المستند بوسم `<!doctype html>` الذي يخبر المتصفح أن يفسر الصفحة وفق لغة HTML الحديثة وليس الإصدارات القديمة المختلفة منها والتي تكون عادةً قبل HTML5.

تمتلك مستندات HTML في الغالب ترويسةً head ومتن body، كما تحتوي الترويسة على معلومات عن المستند؛ أما المتن فيحتوي على النصوص التي في المستند نفسه، وفي هذه الحالة تقول الترويسة أنّ عنوان هذه المستند هو "My home page" وأنها تستخدِم ترميز UTF-8 الذي هو أسلوب نتبعه لترميز نصوص اليونيكود على أساس بيانات ثنائية.

يحتوي متن المستند على ترويسة واحدة، وهي `<h1>` التي تعني الترويسة الأولى، و`<h2>` حتى `<h6>` لتشير إلى ترويسات فرعية، كما يحتوي على فقرتين اثنتين محدَّدتين بوسم `<p>`.

تأتي الوسوم في أشكال كثيرة، فيبدأ العنصر مثل المتن أو الفقرة أو الرابط بوسم افتتاحي مثل `<p>` وينتهي بوسم غالق مثل `</p>`، وقد تكون بعض الوسوم الافتتاحية فيها معلومات أكثر مثل وسم `<a>` في صورة أزواج من `name="value"`، وتسمى هذه المعلومات بالسمات attributes، وفي حالتنا فإن وجهة الرابط توضَّح بـ `href="http://eloquentJavaScript.net"`، وتشير `href` هنا إلى مرجع لنص فائق hypertext reference.

قد لا تغلف بعض الوسوم أيّ شيء ولا تحتاج إلى إغلاقها، ويُعَدّ وسم البيانات الوصفية metadata الذي يأتي بصورة `<meta charset="utf-8">` مثالًا على ذلك.

إذا أردنا كتابة أقواس محددة `<>` داخل نص مستند لتظهر كما هي من غير تفسيرها على أنها محددات وسوم، فيجب كتابتها بصيغة خاصة بها، بحيث يُكتب القوس الابتدائي `<` الذي يمثل علامة "أقل من" في الرياضيات على الصورة `<`، وكذلك القوس الغالق `>` الذي يمثل علامة أكبر من في الرياضيات بصورة `>`.

وهكذا، نكتب محرف `&` (واو العطف اللاتينية ampersand) ثم اسم المحرف أو رمزه ثم فاصلة منقوطة `;` وتُعرف هذه الصيغة باسم الكيان entity، حيث تُستبدَل بالمحرف الذي تعبِّر عنه، وذلك مشابه للطريقة التي تُستخدَم بها الشرطة المائلة العكسية في سلاسل جافاسكربت النصية.

بما أنّ هذه الآلية تعطي معنى خاصًا لمحارف `&`، فإنها بذاتها تحتاج إلى تهريب بأن تُكتب `&` لتظهر بصورتها إذا احتجنا إلى إظهارها في النص، وبالمثل لما بين قيم السمات المغلفة بعلامات تنصيص مزدوجة، إذ يمكن استخدام `"` لإدخال محرف علامة التنصيص ليظهر بصورته.

تُحلَّل HTML بأسلوب يتسامح كثيرًا مع الأخطاء، فإذا كان أحد الوسوم مفقودًا من موضع ما أو نسينا كتابته، فسيكتبه المتصفح ليعالج الخلل، وقد اعتُمِد الأسلوب الذي يحدث به ذلك بحيث تستطيع الاعتماد على جميع المتصفحات الحديثة التي تفعل الشيء نفسه لمعالجة أخطاء الكتابة.

انظر المستند التالي الذي سيعامَل على أنه المستند الذي كتبناه في المثال أعلاه رغم أخطاء الصياغة والبناء التي فيه:

```
<!doctype html>

<meta charset=utf-8>
<title>My home page</title>

<h1>My home page</h1>
<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
  <a href=http://eloquentJavaScript.net>here</a>.
```

عرف المتصفح قد أنّ وسوم <html> و<head> و<body> ليست موجودة، وعرف أنّ الوسمين <meta> و <title> ينتميان إلى الترويسة، والوسم <h1> تعني أنّ متن المستند بدايته من هنا.

كما لم يعد هناك حاجة إلى إغلاق الفقرات بصورة صريحة بما أنّ بدء فقرة جديدة أو إنهاء المستند سيغلقها ضمنيًا، ولعلك لاحظت إن كنت منتبهًا إلى أن علامات التنصيص التي حول قيم السمات ليست موجودة أيضًا.

سنهمِل في هذا الكتاب وسوم <html> و<head> و<body> من الأمثلة للاختصار ولإبقاء الأمثلة بسيطة، لكن سنغلق الوسوم ونضيف علامات التنصيص حول السمات، كما قد نهمِل تصريح doctype وcharset، لكن لا يعني هذا أننا نشجعك على ذلك، فقد تجد المتصفح يتصرف بغرابة ويفعل أشياءً سخيفةً إذا أهملتها، لذلك نريدك أن تتصرف كما لو كانت البيانات الوصفية الخاصة بهما موجودة في الأمثلة حتى لو لم تكن ظاهرة فعلًا في النص أمامك.

13.4 HTML وجافاسكربت

ما يهمنا في HTML فيما يتعلق بهذا الكتاب هو وسم <script>، إذ يسمح لنا بإدخال شيفرة جافاسكربت داخل المستند.

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

ستعمل مثل تلك الشيفرة عندما يرى المتصفح وسم <script> أثناء قراءة HTML، وستُظهر الصفحة نافذة منبثقة حين تُفتح، كما تُشبه دالة alert في المثال أعلاه prompt إلا أنها تخرج نافذة منبثقة تعرض رسالةً ما دون طلب إدخال أي بيانات.

يمثِّل المثال السابق برنامجًا صغير الحجم؛ أما إدخال برامج كبيرة مباشرةً في HTML فهو غير عملي، وبدلًا من ذلك يمكن تزويد وسم <script> بسِمة src لجلب ملف سكربت خارجي من رابط URL ما، وهو ملف نصي يحتوي على برنامج جافاسكربت.

انظر المثال التالي حيث يحتوي ملف code/hello.js على البرنامج السابق نفسه alert("hello!").

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

حين تشير صفحة HTML إلى روابط أخرى لتكون أجزاء منها مثل صورة أو ملف سكربت، فستجلبها المتصفحات مباشرةً أثناء تحميل الصفحة لتعرضها لك مع الصفحة نفسها في الأماكن التي حددها ملف HTML.

يجب إغلاق وسم `<script>` دومًا بوسمه الغالق `</script>` حتى لو كان يشير إلى ملف سكربت لا يحتوي أي شيفرة، فإذا نسيت ذلك، فستُفسَّر بقية الصفحة على أنها جزء من السكربت.

تستطيع تحميل وحدات ES -التي وردت في الفصل العاشر- في المتصفح بتزويد وسم script بسِمة `type="Module"`، كما يمكن أن تعتمد مثل هذه الوحدات على وحدات أخرى باستخدام روابط متعلقة بها على أساس أسماء وحدات في تصريحات `import`.

كذلك يمكن لبعض السمات أن تحتوي على برنامج جافاسكربت، فالوسم `<button>` الذي في المثال التالي والذي سيُظهر زرًا به سمة `onclick`، كما ستعمل قيمة السمة في كل مرة نضغط على الزر فيها.

```
<button onclick="alert('Boom!');">لا تضغط هنا</button>
```

لاحظ أننا اضطررنا لاستخدام علامات تنصيص مفردة للسلسلة النصية التي في سمة `onclick` لأن العلامات المزدوجة كانت مستخدِمة سلفًا للسمة ككل، لكن كان بإمكاني أن أستخدم `"` كذلك.

13.5 داخل صندوق الاختبارات sandbox

لا شك أنّ تشغيل البرامج المحمَّلة من الإنترنت قد يكون خطيرًا، إذ أنك لا تدري مَن وراء أغلب المواقع التي تزورها، وقد يكون فيهم الخبيث واللئيم، كما أنه لا نريد اختراق حواسيبنا بسبب إغفالنا لأمر مثل هذا، لكن الجميل في الموضوع هنا أنك تستطيع تصفح الويب دون الحاجة إلى توكيد الثقة في كل صفحة تزورها، لهذا فإنّ المتصفحات تحدّ كثيرًا من الصلاحيات التي تكون لبرامج جافاسكربت، فلا تستطيع النظر في الملفات التي على حاسوبك أو تعديل أي شيء غير متعلق بالصفحة التي هي فيها.

ويسمى هذا العزل لتلك البرامج بصندوق الرمل أو صندوق الاختبار Sandbox، وإن كانت الترجمة الحرفية له هي صندوق الرمل إلا أنّ وظيفته هي عزل البرمجيات التي لا نثق فيها إلى حين البَت في أمرها أو تنتفي الحاجة إليها، وتعمل هذه البرامج كما تشاء داخل هذا الصندوق دون أذى للحاسوب نفسه أو لبياناتك، فلو كان هذا الصندوق قفصًا حديديًا سميك الجدران، فستكون البرامج حينها هي الكائنات التي تعبث فيه كما تشاء دون أن تؤذي من هو خارج القفص، ودون أن تستطيع هي نفسها الخروج.

لكن الأمر الصعب عند التعامل مع هذا الصندوق هو السماح لهذه البرامج بفسحة تكفي لتكون مفيدةً مع حجزها عن التسبب في أذى للحاسوب والبيانات التي عليه في الوقت نفسه، لكن احذر أن تشمل هذه الفسحة أمورًا مثل التواصل مع الخوادم الأخرى وقراءة محتويات حافظة النسخ واللصق، فهنا قد يتطور الأمر إلى منحى خطير على الخصوصية.

قد يخرج علينا أحد بين الحين والآخر بطريقة جديدة للتحايل على حدود المتصفح والتسبب في اختراق وأذى للمستخدم من تسريب معلومات طفيفة عنه حتى السيطرة التامة على الجهاز المخترَق، كما يستجيب مطورو المتصفحات لهذه التهديدات بإصلاح الثغرات التي تتسبب فيها إلى حين ظهور مشكلة جديدة، لكن مكمن

الخطر هنا هو عدم إدراك المطورين للمشكلة إذا لم تُفضَح وتنتشر، فإذا استغلها أحد أو شركة أو منظمة أو جهة حكومية دون الإبلاغ عنها فستكون هنا الخسائر المحتملة أكبر.

13.6 التوافقية وحروب المتصفحات

كان المتصفح المسيطر على الساحة في بدايات الويب يسمى موزايك Mosaic، ثم انتقل السوق إلى متصفح نِت سكيب Netscape الذي ما لبث أن أفرغ الساحة قهرًا لمتصفح إنترنت إكسبلورر Internet Explorer الشهير الخاص بمايكروسوفت.

كانت الجهة التي تقف خلف المتصفح في كل مرة تشعر أنّ لها الحق في اختراع المزايا الجديدة التي تشاء للويب، وبما أن أغلب المستخدِمين كانوا يستخدِمون متصفحها، فستضطر المواقع إلى استخدام تلك المزايا مع غض النظر عن باقي المتصفحات، وهذا سبب الكثير من مشاكل التوافقية للمواقع مع المتصفحات المختلفة. نستطيع القول أنّ هذا كان هو العصر المظلم للتوافقية، والذي كان يُعرف بعصر حروب المتصفحات، كما كان المطورون يجدون أنفسهم بين منصتين أو ثلاث لا تتوافق أي منها مع الأخرى بدلًا من ويب واحدة، وزاد الطين بلة امتلاء المتصفحات التي كانت في السوق في بداية الألفية الثالثة -نحو 2003- بالزلات البرمجية Bugs والمشاكل، وكل متصفح له مشاكله، وهكذا كانت حياة المطورين أشبه بالحرب فعليًا.

ظهر بعد ذلك متصفح موزيلا فاير فوكس Mozilla Firefox، وهو فرع لا يهدف للربح من نِت سكيب ليبارز إنترنت إكسبلورر في السوق في أواخر عام 2002، وقد سيطر على حصة كبيرة من السوق لعدم اهتمام مايكروسوفت بمتابعة التطوير وزهدها في مجال المتصفحات حينها، ثم خرجت جوجل بمتصفح خاص بها هو جوجل كروم Chrome، وكذلك شركة Apple بمتصفح سفاري Safari الذي اكتسب شهرةً بسبب أنه يأتي مع أجهزتها تلقائيًا، وبالتالي صار لدينا أربعة متصفحات رئيسية على الساحة بدلًا من واحد فقط.

اتجهت هذه الهيئات الأربعة نحو وضع معايير موحَّدة وأساليب هندسية أفضل لتصميم هذه المتصفحات، لتعطينا متصفحات لا تعاني كثيرًا من الزلات البرمجية ولا مشاكل التوافقية، وقد رأت مايكروسوفت مؤخرًا أنّ حصتها في هذا السوق قد تهاوت كثيرًا، فتبنّت هذا الرأي في النهاية في متصفح إيدج Edge الخاص بها واستبدلت به متصفحها القديم.

يصب كل هذا في مصلحة من يقرِّر تعلم تطوير الويب هذه الأيام، إذ أن الإصدارات الأخيرة من هذه المتصفحات تكاد تتصرف بالأسلوب نفسه تقريبًا وليس لديها الكثير من الزلات البرمجية.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

14. نموذج كائن المستند DOM

التخطيط قبل العمل يقلل مرات الفشل.

حين تفتح صفحة ويب في متصفحك فإن المتصفح يجلب نص HTML الخاص بها ويحلله كما يفعل المحلل parser الذي أنشأناه في الفصل الثاني عشر مع البرامج، كما يبني المتصفح نموذجًا لهيكل المستند ويستخدم هذا النموذج لإظهار الصفحة كما تراها على الشاشة.

يُعَدّ هذا التمثيل للمستند طريقةً في صناديق الاختبار sandboxes التي في برامج جافاسكربت، وهو هيكل بيانات يمكنك قراءته وتعديله، كما يتصرف على أساس هيكل بيانات حي تتغير الصفحة بتعديله لتعكس هذه التغييرات.

14.1 هيكل المستند

تخيَّل مستند HTML أنه مجموعة متشعِّبة من الصناديق، وتغلِّف فيها وسوم مثل `<body>` و`</body>` وسومًا أخرى، وهذه الوسوم تحتوي بدورها على وسوم أو نصوص أخرى.

انظر هذا المثال من الفصل السابق: جافاسكربت والمتصفحات.

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
```

```
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentJavaScript.net">here</a>.</p>
  </body>
</html>
```

ستظهر الصفحة التي في هذا المثال كما يلي:

يتبع هيكل البيانات الذي يستخدمه المتصفح لتمثيل هذا المستند هذا الشكل، فهناك كائن لكل صندوق يمكننا التفاعل معه لمعرفة أشياء، مثل وسم HTML التي يمثلها والصناديق والنصوص التي يحتوي عليها، ويسمى هذا التمثيل بنموذج كائن المستند Document Object Model أو DOM اختصارًا.

وتعطينا رابطة `document` العامة وصولًا إلى هذه الكائنات، وتشير خاصية `documentElement` إلى الكائن الذي يمثل وسم `<html>`، وبما أيّ مستند HTML فيه ترويسة Head ومتن Body، فسيحتوي على خاصيتي `head` و`body` اللتين تشيران إلى هذين العنصرين أيضًا.

14.2 الأشجار

لو أنك تذكر أشجار البُنى syntax trees التي تحدثنا عنها في الفصل الثاني عشر والتي تشبه هياكلها هياكل المستندات التي في المتصفح شبهًا كبيرا؛ فكل عُقدة node تشير إلى عُقد أخرى وقد يكون

للفروع children فروعًا أخرى، وهذا الشكل هو نموذج للهياكل المتشعبة حيث تحتوي العناصر على عناصر فرعية تشبهها.

نقول على هيكل البيانات أنه شجرة إذا احتوي على هيكل ذي بنية متفرعة branching وليس فيه دورات cycles -بحيث لا يمكن للعُقدة أن تحتوي نفسها مباشرةً أو بصورة غير مباشرة-، وله جذر واحد معرَّف جيدًا، وهذا الجذر في حالة DOM هو `document.documentElement`.

نتعرض للأشجار كثيرًا في علوم الحاسوب، فهي تُستخدَم للحفاظ على مجموعات مرتبة من البيانات، حيث يكون إدخال العناصر أو العثور عليها أسهل داخل شجرة من لو كان في مصفوفة مسطحة، وذلك إضافة إلى استخدامات أخرى مثل تمثيل الهياكل التعاودية recursive structures مثل مستندات HTML أو البرامج.

تمتلك الشجرة عدة أنواع مختلفة من العُقد، فشجرة البُنى للغة Egg التي أنشأناها في الفصل الثاني عشر من هذا الكتاب كان لها معرِّفات identifiers وقيم values وعُقد تطبيقات application nodes، كما يمكن أن يكون لعُقد التطبيقات تلك فروعًا، في حين يكون للمعرِّفات وللقيم أوراقًا leaves أو عُقدًا دون فروع.

ينطبق المنطق نفسه على DOM، فعُقد العناصر التي تمثِّل وسوم HTML تحدِّد هيكل المستند، ويمكن أن يكون لها عُقدًا فرعيةً child nodes، وأحد الأمثلة على تلك العُقد هو `document.body`. كذلك فإن تلك العُقد الفرعية قد تكون عُقدًا ورقيةً leaf nodes مثل النصوص أو عُقد التعليقات comment nodes.

يملك كل كائن عُقدة في DOM خاصية `nodeType` تحتوي على رمز -أو عدد- يعرِّف نوع العُقدة، فتحمل العناصر الرمز 1 الذي يُعرَّف أيضًا على أساس خاصية ثابتة لـ `Node.ELEMENT_NODE`؛ أما عُقد النصوص التي تمثِّل أجزاءً من النصوص داخل المستند فتحصل على الرمز 3 وهو `Node.TEXT_NODE`، في حين تحمل التعليقات الرمز 8 الذي هو `Node.COMMENT_NODE`.

يوضِّح الشكل التالي شجرة مستندنا بصورة أفضل:

العُقد النصية هنا هي الأوراق، والأسهم توضح علاقة الأصل والفرع بين العُقد.

14.3 المعيار

لا يتلاءم استخدام رموز عددية مبهمة لتمثيل أنواع العُقد مع طبيعة جافاسكربت، وسنرى في هذا الفصل أجزاءً أخرى من واجهة DOM ستبدو متعِبة ومستهجنة، وذلك لأن DOM لم يصمَّم من أجل جافاسكربت وحدها، بل يحاول أن يكون واجهة غير مرتبطة بلغة بعينها ليُستخدم في أنظمة أخرى، فلا يكون من أجل HTML وحدها بل لـ XML كذلك، وهي صيغة بيانات عامة لها بنية تشبه HTML.

لكن مزية المعيارية هنا ليست مقنعة ولا مبررة، فالواجهة التي تتكامل تكاملًا حسنًا مع اللغة التي تستخدمها ستوفر عليك وقتًا موازنة بالواجهة التي تكون موحدة على اختلاف اللغات، وانظر خاصية `childNodes` التي في عُقد العناصر في DOM لتكون مثالًا على هذا التكامل السيء، فتلك الخاصية تحمل كائنًا شبيهًا بالمصفوفة object array-like مع خاصية `length` وخصائص معنونة بأعداد للوصول إلى العُقد الفرعية، لكنه نسخة instance من النوع `NodeList` وليس مصفوفةً حقيقيةً، لذا فليس لديه توابع مثل `slice` و`map`.

ثم هناك مشاكل ليس لها مراد إلا سوء التصميم، فليست هناك مثلًا طريقةً لإنشاء عقدة جديدة وإضافة فروع أو سمات إليها، بل يجب عليك إنشاء العُقدة ثم إضافة الفروع والسمات واحدة واحدة باستخدام الآثار الجانبية side effects، وعلى ذلك ستكون الشيفرة التي تتعامل مع DOM طويلةً ومتكررةً وقبيحةً أيضًا.

لكن هذه المشاكل والعيوب ليست حتميةً، فمن الممكن تصميم طرق مطوَّرة وأفضل للتعبير عن العمليات التي تنفذها أنت طالما تسمح لنا جافاسكربت بإنشاء تجريداتنا الخاصة، كما تأتي العديد من المكتبات الموجهة للبرمجة للمتصفحات بمثل تلك الأدوات.

14.4 التنقل داخل الشجرة

تحتوي عُقد DOM على روابط link كثيرة جدًا تشير إلى العُقد المجاورة لها، انظر المخطط التالي مثلًا:

رغم أن المخطط لا يظهر إلا رابطًا واحدًا من كل نوع إلا أنّ كل عُقدة لها خاصية `parentNode` التي تشير إلى العُقدة التي هي جزء منها إن وجدت، وبالمثل فكل عُقدة عنصر -التي تحمل النوع 1- لها خاصية `childNodes` التي تشير إلى كائن شبيه بالمصفوفة يحمل فروعه.

تستطيع نظريًا التحرك في أي مكان داخل الشجرة باستخدام روابط الأصول والفروع هذه، لكن جافاسكربت تعطيك أيضًا وصولًا إلى عدد من الروابط الإضافية الأخرى، فتشير الخاصيتان `firstChild` و`lastChild` إلى العنصرين الفرعيين الأول والأخير، أو تكون لهما القيمة `null` للعُقد التي ليس لها فروع، وبالمثل أيضًا تشير `previousSibling` و`nextSibling` إلى العُقد المتجاورة، وهي العُقد التي لها الأصل نفسه أو الأصل الذي يظهر قبل أو بعد العُقدة مباشرةً، وستحمل `previousSibling` القيمة `null` لأول فرع لعدم وجود شيء قبله، وكذلك ستحمل `nextSibling` القيمة `null` لآخر فرع.

لدينا أيضًا الخاصية `children` التي تشبه `childNodes` لكن لا تحتوي إلا عناصر فرعية -أي ذات النوع 1- ولا شيء آخر من بقية أنواع العُقد الفرعية، وذلك مفيد إذا لم تكن تريد العُقد النصية.

نفضِّل استخدام الدوال التعاودية recursive functions عند التعامل مع هيكل بيانات متشعب كما في المثال أدناه، حيث تقرأ الدالة التالية المستند بحثًا عن العُقد النصية التي تحتوي على نص معطى وتُعيد `true` إذا وجدته:

```
function talksAbout(node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let child of node.childNodes) {
      if (talksAbout(child, string)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

تحمل الخاصية `nodeValue` للعُقدة النصية السلسلة النصية التي تمثلها.

14.5 البحث عن العناصر

رغم أنّ التنقل بين الروابط سابقة الذكر يصلح بين الأصول parents والفروع children والأشقاء siblings، إلا أننا سنواجه مشاكل إذا أردنا البحث عن عُقدة بعينها في المستند.

فمن السيئ اتباع الطريق المعتاد من `document.body` عبر مسار ثابت من الخصائص، إذ يسمح هذا بوضع فرضيات كثيرة في برنامجنا عن الهيكل الدقيق للمستند، وهو الهيكل الذي قد تريد تغييره فيما بعد.

تُنشأ كذلك العُقد النصية للمسافات الفارغة بين العُقد الأخرى، فوسم `<body>` يحمل أكثر من ثلاثة فروع والذين هم عنصر `<h1>` وعنصرين `<p>`، وإنما المسافات الفارغة بينها وقبلها وبعدها أيضًا، وبالتالي يكون سبعة فروع.

إذا أردنا الوصول إلى سمة `href` للرابط الذي في ذلك المستند فلن نكتب "اجلب الفرع الثاني للفرع السادس من متن المستند"، بل الأفضل هو قول "اجلب الرابط الأول في المستند"، ونحن نستطيع فعل ذلك، انظر كما يلي:

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

تحتوي جميع عُقد العناصر على التابع `getElementsByTagName` الذي يجمع العناصر التي تحمل اسم وسم ما، وتكون منحدرة -فروعًا مباشرةً أو غير مباشرة- من تلك العُقدة ويُعيدها على أساس كائن شبيه بالمصفوفة.

لإيجاد عُقدة منفردة بعينها، أعطها سمة `id` واستخدم `document.getElementById`، أي كما يلي:

```
<p>My ostrich Gertrude:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

هناك تابع ثالث شبيه بما سبق هو `getElementsByClassName` يبحث في محتويات عُقدة العنصر مثل `getElementsByTagName` ويجلب جميع العناصر التي لها السلسلة النصية المعطاة في سمة `class`.

14.6 تغيير المستند

يمكن تغيير كل شيء تقريبًا في هيكل البيانات الخاص ب DOM، إذ يمكن تعديل شكل شجرة المستند من خلال تغيير علاقات الأصول والفروع.

تحتوي العُقد على التابع `remove` لإزالتها من عُقدة أباها، ولكي تضيف عُقدة فرعية إلى عُقدة عنصر node element فيمكننا استخدام `appendChild` التي تضعها في نهاية قائمة الفروع، أو `insertBefore` التي تدخِل العُقدة المعطاة على أساس أول وسيط argument قبل العُقدة المعطاة على أساس وسيط ثاني.

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

لا يمكن للعُقدة أن توجد في المستند إلا في مكان واحد فقط، وعليه فإنّ إدخال فقرة Three في مقدمة الفقرة One سيزيلها أولًا من نهاية المستند ثم يدخلها في أوله، لنحصل على Three|One|Two، وبناءً على ذلك ستتسبب جميع العمليات التي تدخل عُقدة في مكان ما -على أساس أثر جانبي- في إزالتها من موقعها الحالي إن كان لها واحد.

يُستخدَم التابع `replaceChild` لاستبدال عُقدة فرعية بأخرى، ويأخذ عُقدتين على أساس وسيطين، واحدة جديدة والعُقدة التي يراد تغييرها، ويجب أن تكون العُقدة المراد تغييرها عُقدة فرعية من العنصر الذي استُدعي عليه التابع، لاحظ أنّ كلًا من `replaceChild` و`insertBefore` تتوقعان العُقدة الجديدة على أساس وسيط أول لهما.

14.7 إنشاء العقد

لنقل أنك تريد كتابة سكربت يستبدل جميع الصور -أي وسوم `<img>`- في المستند ويضع مكانها نصوصًا في سمات `alt` لها، والتي تحدِّد نصًا بديلًا عن الصور، حيث سيحذف الصور وسيضيف عُقدًا نصيةً جديدةً لتحل محلها، كما ستُنشأ العُقد النصية باستخدام تابع `document.createTextNode` كما يلي:

```
<p>The <img src="img/cat.png" alt="Cat"> in the
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

إذا كان لدينا سلسلة نصية، فستعطينا `createTextNode` عُقدةً نصية نستطيع إدخالها إلى المستند لنجعلها تظهر على الشاشة، وستبدأ الحلقة التكرارية التي ستمر على الصور من نهاية القائمة، لأن قائمة العُقد التي أعادها تابع مثل `getElementsByTagName` -أو سمة مثل `childNodes`- هي قائمة حية بمعنى أنها تتغير كلما تغير المستند، وإذا بدأنا من المقدمة وحذفنا أول صورة فسنُفقِد القائمة أول عناصرها كي تتكرر الحلقة التكرارية الثانية، حيث `i` تساوي 1، وستتوقف لأن طول المجموعة الآن صار 1 كذلك.

أما إذا أردت تجميعة ثابتة solid collection من العُقد -على النقيض من العُقد الحية- فستستطيع تحويل التجميعة إلى مصفوفة حقيقية باستدعاء `Array.from` كما يلي:

```
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

استخدم التابع `document.createElement` لإنشاء عُقد عناصر، حيث يأخذ هذا التابع اسم الوسم ويعيد عُقدةً جديدةً فارغةً من النوع المعطى.

انظر المثال التالي الذي يعرِّف الأداة `elt` التي تنشئ عُقدة عنصر وتعامل بقية وسائطها على أساس فروع لها، ثم تُستخدَم هذه الدالة لإضافة خصائص إلى اقتباس نصي، أي كما يلي:

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child != "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", preface to the second edition of ",
        elt("em", "The Open Society and Its Enemies"),
        ", 1950"));
</script>
```

14.8 السمات Attributes

يمكن الوصول إلى بعض سمات العناصر مثل `href` الخاصة بالروابط من خلال خاصية الاسم نفسه على كائن DOM الخاص بالعنصر وهذا شأن أغلب السمات القياسية المستخدَمة، لكن تسمح لك HTML بإسناد set أيّ عدد من السمات إلى العُقد، وذلك مفيد لأنه يسمح لك بتخزين معلومات إضافية في المستند، فإذا ألّفت أسماء سمات خاصة بك فلن تكون موجودة على أساس خصائص في عُقدة العنصر، بل يجب أن تستخدِم التابعَين `getAttribute` و `setAttribute` لكي تتعامل معها.

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
```

```
  let paras = document.body.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

يفضَّل أن تسبق أسماء هذه السمات التي تنشئها أنت بـ `data-` كي تتأكد أنها لن تتعارض مع أي سمة أخرى. فمثلًا، لدينا سمة `class` شائعة الاستخدام وهي كلمة مفتاحية في لغة جافاسكربت، كما كانت بعض تطبيقات جافاسكربت القديمة -لأسباب تاريخية- لا تستطيع التعامل مع أسماء الخصائص التي تطابق كلمات مفتاحية، وقد كانت الخاصية التي تُستخدَم للوصول إلى هذه السمة هي `className`، لكن تستطيع الوصول إليها تحت اسمها الحقيقي `"class"` باستخدام التابعَين `getAttribute` و`setAttribute`.

14.9 مخطط المستند Layout

لعلك لاحظت أن الأنواع المختلفة من العناصر توضع بتخطيط مختلف، فبعضها -مثل الفقرات `<p>` أو الترويسات `<h1>`- يأخذ عرض المستند بأكمله وتُخرَج على أسطر مستقلة، وتسمى هذه العناصر بالعناصر الكتلية block elements؛ في حين بعضها الآخر مثل الروابط `<a>` والخط السميك `<strong>` تُخرَج على السطر نفسه مع النص المحيط بها، وتسمى هذه العناصر بـ: العناصر السطرية inline elements.

يستطيع المتصفح أن يضع مخططًا لأي مستند، بحيث يعطي كل عنصر فيه حجمًا وموضعًا وفقًا لنوعه ومحتواه، بعدها يُستخدَم هذا المخطط لرسم المستند في ما يعرضه المتصفح.

يمكن الوصول إلى حجم وموضع أي عنصر من خلال جافاسكربت، إذ تعطيك الخاصيتان `offsetWidth` و`offsetHeight` المساحة التي تأخذها العناصر مقاسة بالبكسلات pixels، وتُعَدّ البكسل أصغر وحدة قياس في المتصفح، وقد كانت تساوي أصغر نقطة تستطيع الشاشة عرضها، لكن الشاشات الحديثة التي تستطيع رسم نقاط صغيرة للغاية لا ينطبق عليها هذا المقياس، حيث يساوي البكسل الواحد عدة نقاط فيها، وبالمثل تعطي `clientWidth` و`clientHeight` حجم المساحة داخل العنصر متجاهلة عرض الإطار.

```
<p style="border: 3px solid red">
  أنا موجود داخل إطار
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
```

```
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

أفضل طريقة لمعرفة الموضع الدقيق لأي عنصر على الشاشة هي باستخدام التابع `getBoundingClientRect`، حيث يُعيد كائنًا فيه خصائص `top` و`bottom` و`left` و`right`، مشيرةً إلى مواضع البكسلات لجوانب العنصر نسبةً إلى أعلى يسار الشاشة، فإذا أردتها منسوبةً إلى المستند كله، فيجب إضافة موقع التمرير الحالي في المستند والذي ستجده في الرابطتين `pageXoffset` و`pageYoffset`.

قد يكون تخطيط المستند مجهدًا لكثرة تفاصيله، لذا لا تعيد محركات المتصفحات إعادة تخطيط المستند في كل مرة تغيره بل تنتظر أطول فترة ممكنة، فحين ينتهي برنامج جافاسكربت من تعديل مستند، فسيكون على المتصفح أن يحسب تخطيطًا جديدًا لرسم المستند الجديد على الشاشة. كذلك إذا طلب برنامج ما موضع أو حجم شيء من خلال قراءة خاصية مثل `offsetHeight` أو استدعاء `getBoundingClientRect`، ذلك أن توفير المعلومات الصحيحة يتطلب حساب التخطيط.

أما إذا كان البرنامج ينتقل بين قراءة معلومات مخطط DOM وتغيير DOM، فسيتطلب الكثير من حسابات التخطيط وعليه سيكون بطيئًا جدًا، كما تُعَدّ الشيفرة التالية مثالًا على ذلك، إذ تحتوي على برنامجين مختلفين يبنيان سطرًا من محارف X بعرض 2000 بكسل، ويقيسان الوقت الذي يستغرقه كل واحد منهما.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Current time in milliseconds
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms
```

```
  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

14.10 التنسيق Styling

رأينا أنّ عناصر HTML المختلفة تُعرَض على الشاشة بطرق مختلفة، فبعضها يُعرَض في كتل مستقلة، وبعضها يكون داخل السطر نفسه، كما يضاف تخصيص مثل `<strong>` إلى بعض النصوص لجعلها سميكة، وكذلك يُضاف `<a>` إلى بعضها الآخر كي تظهر بلون أزرق وتحتها خط دلالةً على كونها رابطًا تشعبيًا.

ترتبط الطريقة التي يعرض بها وسم `<img>` صورة ما، أو يجعل وسم `<a>` رابطًا يذهب إلى صفحة أخرى عند النقر عليه ارتباطًا وثيقًا إلى نوع العنصر، لكن نستطيع تغيير التنسيق المرتبط بالعنصر مثل لون النص أو وضع خط أسفله.

انظر المثال التالي على استخدام خاصية `style`:

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

يمكن لسمة التنسيق style attribute أن تحتوي تصريحًا واحدًا أو أكثر، وهو خاصية -مثل `color`- متبوعة بنقطتين رأسيتين وقيمة -مثل `green` في المثال أعلاه-، وإذا كان لدينا أكثر من تصريح واحد فيجب فصل التصريحات بفواصل منقوطة كما في `"color: red; border: none"`.

يتحكم التنسيق كما ترى في جوانب كثيرة من المستند. فمثلًا، تتحكم خاصية `display` في عرض العنصر على أنه كتلة مستقلة أو عنصر سطري، أي كما يلي:

```
يُعرض هذا النص <strong>في السطر كما ترى</strong>,
<strong style="display: block">مثل كتلة</strong>, و
<strong style="display: none">لا يُعرض على الشاشة</strong>.
```

سيُعرض وسم `block` في المثال السابق في سطر منفصل بما أن عناصر الكتل لا تُعرض داخل سطر مع نصوص حولها؛ أما الوسم الأخير فلن يُعرض مطلقًا بسبب `none` التي تمنع العنصر من الظهور على الشاشة،

وتلك طريقة لإخفاء العناصر وهي مفضَّلة على الحذف النهائي من المستند لاحتمال الحاجة إليها في وقت لاحق.

يمكن لشيفرة جافاسكربت أن تعدّل مباشرةً على تنسيق عنصر ما من خلال خاصية `style` لذلك العنصر، وهذه الخاصية تحمل كائنًا له خصائص لكل خصائص التنسيق المحتملة، كما تكون قيم هذه الخصائص سلاسل نصية نكتبها كي نغيِّر جزءًا بعينه من تنسيق العنصر.

```
<p id="para" style="color: purple">
  هذا نص جميل
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

تحتوي بعض أسماء خصائص التنسيقات على شرطة - مثل `font-family`، وبما أنّ أسماء هذه الخصائص يصعب التعامل معها في جافاسكربت إذ يجب كتابة `style["font-family"]`، فإن الأسماء التي في كائن `style` لتلك الخصائص تُحذف منها الشُّرَط التي فيها وتُجعل الأحرف التي بعدها أحرف كبيرة كما في `style.fontFamily`.

14.11 التنسيقات الانسيابية Cascading Styles

يسمى نظام تصميم وعرض العناصر في HTML باسم CSS، وهي اختصار Cascading Style Sheets أو صفحات التنسيقات الانسيابية، وتُعَدّ صفحة التنسيق style sheet مجموعةً من القوانين التي تحكم مظهر العناصر في مستند ما، ويمكن كتابتها داخل وسم `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>الآن <strong>النص السميك</strong> صار مائلًا ورماديًا.</p>
```

وتشير الانسيابية التي في هذه التسمية إلى إمكانية جمع عدة قواعد معًا وانسيابها من الأب للابن لإنتاج التنسيق النهائي لعنصر ما.

تعطَّل أثر التنسيق الافتراضي لوسوم `<strong>` في المثال السابق التي تجعل الخط سميكًا بسبب القاعدة الموجودة في وسم `<style>` التي تضيف تنسيق الخط `font-style` ولونه `color`.

وإذا عرَّفت عدة قواعد قيمةً لنفس الخاصية، فإن أحدث قاعدة قُرِئت ستحصل على أسبقية أعلى وتفوز، لذا فإذا كان وسم `<style>` يحتوي على `font-weight: normal` وعارض قاعدة `font-weight` الافتراضية، فسيكون النص عاديًا وليس سميكًا، فالتنسيقات التي في سمة `style` والتي تُطبَّق مباشرةً على العُقدة لها أولوية أعلى وتكون هي الفائزة دائمًا.

من الممكن استهداف أشياء غير أسماء الوسوم في قواعد CSS، إذ ستُطبَّق قاعدة موجهة لـ `.abc` على جميع العناصر التي فيها `"abc"` في سمة `class` الخاصة بها، وكذلك قاعدة لـ `#xyz` ستُطبق على عنصر له سمة `id` بها `"xyz"`، والتي يجب أن تكون فريدةً ولا تتكرر في المستند.

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}
```

لا تنطبق قاعدة الأولوية التي تفضِّل أحدث قاعدة معرَّفة إلا حين تكون جميع القواعد لها النوعية specificity نفسها، ونوعية القاعدة مقياس لدقة وصف العناصر المتطابقة، وتُحدَّد بعدد جوانب العنصر التي يتطلبها ونوعها -أي الوسم أو الصنف أو المعرِّف ID. فمثلًا، تكون القاعدة التي تستهدف `p.a` أكثر تحديدًا من قاعدة تستهدف `p` أو `.a` فقط، وعليه تكون لها الأولوية.

تطبّق الصيغة `p > a {...}` التنسيقات المعطاة على جميع وسوم `<a>` التي تكون فروعًا مباشرةً من وسوم `<p>`، وبالمثل تطبّق `p a {...}` على جميع وسوم `<a>` الموجودة داخل وسوم `<p>` سواءً كانت فروعًا مباشرةً أو غير مباشرة.

14.12 محددات الاستعلامات Query Selectors

لن نستخدم صفحات التنسيقات كثيرًا في هذا الكتاب، إذ يحتاج تعقيدها وتفصيلها إلى كتاب خاص بها، لكن فهمها ينفعك عند البرمجة في المتصفح، والسبب الذي جعلنا نذكر بُنية المحدِّد هنا -وهي الصيغة المستخدَمة في صفحات التنسيقات لتحديد العناصر التي تنطبق عليها مجموعة تنسيقات بعينها- هو أننا نستطيع استخدام التركيب اللغوي نفسه على أساس طريقة فعالة للعثور على عناصر DOM.

يأخذ التابع `querySelectorAll` المعرَّف في كائن `document` وفي عُقد العناصر، ويأخذ سلسلةً نصيةً لمحدِّد ويُعيد `NodeList` تحتوي جميع العناصر المطابقة.

```
<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // All <p> elements
  // → 4
  console.log(count(".animal"));     // Class animal
  // → 2
  console.log(count("p .animal"));   // Animal inside of <p>
  // → 2
  console.log(count("p > .animal")); // Direct child of <p>
  // → 1
</script>
```

لا يكون الكائن المعاد من `querySelectorAll` حيًّا على عكس توابع مثل `getElementsByTagName`، كما لن يتغير إذا غيرت المستند، إذ لا يزال مصفوفةً غير حقيقية، لذا ستحتاج إلى استدعاء `Array.from` إذا أردت معاملته على أنه مصفوفة.

يعمل التابع `querySelector` -دون `All`- بأسلوب مشابه، وهو مفيد إذا أردت عنصرًا منفردًا بعينه، إذ سيعيد أول عنصر مطابق أو `null` إذا لم يكن ثمة مطابقة.

14.13 التموضع والتحريك

تؤثر خاصية التنسيق position على شكل التخطيط تأثيرًا كبيرًا، ولها قيمة static افتراضيًا، أي أن العنصر يظل في موضعه العادي في المستند، وحين تُضبط على relative فسيأخذ مساحةً في المستند أيضًا لكن مع اختلاف أنّ الخصائص التنسيقية top وleft يمكن استخدامها لتحريكه نسبة إلى ذلك الموضع العادي له.

أما حين تُضبط position على absolute فسيُحذَف العنصر من التدفق الاعتيادي للمستند normal flow، أي لا يأخذ مساحة، وإنما قد يتداخل مع عناصر أخرى، وتُستخدم top وleft هذه المرة لموضعة العنصر بصورة مطلقة هذه المرة نسبةً إلى الركن الأيسر العلوي لأقرب عنصر مغلِّف تكون خاصية position له غير static، أو نسبة إلى المستند ككل إن لم يوجد عنصر مغلِّف.

نستخدِم ما سبق عند إنشاء تحريك animation، كما يوضح المستند التالي الذي يعرض صورة قطة تتحرك في مسار قطع ناقص ellipse.

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

تكون صورتنا في منتصف الصفحة ونضبط خاصية position لتكون relative، وسنحدِّث تنسيقي الصورة top وleft باستمرار من أجل تحريك الصورة.

تستخدِم السكربت requestAnimationFrame لجدولة دالة animate كي تعمل كلما كان المتصفح جاهزًا لإعادة رسم الشاشة أو تغيير المعروض عليها، وتستدعي دالة animate نفسها

requestAnimationFrame لجدولة التحديث التالي، وحين تكون نافذة المتصفح كلها نشطةً أو نافذة اللسان (تبويب) فقط، فإن ذلك يتسبب في جعل معدل التحديثات نحو 60 تحديثًا في الثانية، مما يجعل مظهر العرض ناعمًا وجميلًا، فإذا حدَّثنا DOM في حلقة تكرارية فستتجمد الصفحة ولن يظهر شيء على الشاشة، إذ لا تحدِّث المتصفحات العرض الخاص بها أثناء تشغيل برنامج جافاسكربت ولا تسمح لأيّ تفاعل مع الصفحة، من أجل ذلك نحتاج إلى requestAnimationFrame، إذ تسمح للمتصفح أن يعرف أننا انتهينا من هذه المرحلة، ويستطيع متابعة فعل المهام الخاصة بالمتصفحات، مثل تحديث الشاشة والتجاوب مع تفاعل المستخدم.

يُمرَّر الوقت الحالي إلى دالة التحريك على أساس وسيط، ولكي نضمن أن حركة القطة ثابتة لكل ميلي ثانية، فإنها تبني السرعة التي تتغير بها الزاوية على الفرق بين الوقت الحالي وبين آخر وقت عملت فيه الدالة، فإذا حرَّكتَ الزاوية بمقدار ثابت لكل خطوة، فستبدو الحركة متعثرةً وغير ناعمة إذا كان لدينا مهمة أخرى كبيرة تعمل على نفس الحاسوب مثلًا، وتمنع الدالة من العمل حتى ولو كانت فترة المنع تلك جزء من الثانية.

تُنفَّذ الحركة الدائرية باستخدام دوال حساب المثلثات Math.cos وMath.sin، كما سنشرح هذه الدوال إذا لم يكن لك بها خبرة سابقة بما أننا سنستخدمها بضعة مرات في هذا الكتاب.

تُستخدَم هاتان الدالتان لإيجاد نقاط تقع على دائرة حول نقطة الإحداثي الصفري (0,0) والتي لها نصف قطر يساوي 1، كما تفسِّران وسيطها على أساس موضع على هذه الدائرة مع صفر يشير إلى النقطة التي على أقصى يمين الدائرة، ويتحرك باتجاه عقارب الساعة حتى يقطع محيطها الذي يساوي 2 باي -أي 2π- والتي يكون مقدارها هنا 6.28 تقريبًا.

تخبرك Math.cos بإحداثية x للنقطة الموافقة للموضع الحالي، في حين تخبرك Math.sin بإحداثية y، وأيّ موضع أو زاوية أكبر من 2 باي 2π -أي محيط الدائرة- أو أقل من صفر يكون صالحًا ومقبولًا، ويتكرر الدوران إلى أن تشير a+2π إلى نفس الزاوية التي تشير إليها a.

تسمى هذه الوحدة التي تقاس بها الزوايا باسم الزاوية نصف القطرية أو راديان radian، والدائرة الكاملة تحتوي على 2 π راديان، ويمكن الحصول على الثابت الطبيعي باي π في جافاسكربت من خلال Math.PI.

يكون لشيفرة تحريك القطة مقياسًا يدعى angle للزاوية الحالية التي عليها التحريك، بحيث يتزايد في كل مرة تُستدعى فيها دالة animate، ثم يمكن استخدام هذه الزاوية لحساب الموضع الحالي لعنصر الصورة.

يُحسب التنسيق العلوي `top` باستخدام `Math.sin` ويُضرب في 20، وهو نصف القطر الرأسي للقطع الناقصة في مثالنا، وبالمثل يُبنى تنسيق `left` على `Math.cos`، ويُضرب في 200 لأن القطع الناقص عرضه أكبر من ارتفاعه.

لاحظ أن التنسيقات تحتاج إلى وحدات في الغالب، وفي تلك الحالة فإننا نحتاج أن نلحق `"px"` إلى العدد ليخبر المتصفح أن وحدة العدّ التي نستخدمها هي البكسل -وليس سنتيمترات أو ems أو أي شيء آخر- وهذه النقطة مهمة لسهولة نسيانها، حيث ستتسبب كتابة أعداد دون وحدات في تجاهل التنسيق الخاص بك، إلا إن كان العدد صفرًا، وذلك لأن معناه لا يختلف مهما اختلفت الوحدات.

14.14 خاتمة

تستطيع البرامج المكتوبة بجافاسكربت فحص المستند الذي يعرضه المتصفح والتدخل فيه بالتعديل، من خلال هيكل بيانات يسمى DOM، حيث يمثِّل هذا الهيكل نموذج المتصفح للمستند، ويعدّله برنامج جافاسكربت من أجل التعديل في المستند المعروض على الشاشة.

يُنظَّم DOM في هيئة شجرية، بحيث تُرتَّب العناصر فيها هرميًا وفقًا لهيكل المستند، والكائنات التي تمثل العناصر لها خصائص مثل `parentNode` و`childNodes` التي يمكن استخدامها للتنقل في الشجرة، كما يمكن التأثير على طريقة عرض المستند من خلال التنسيقات، إما عبر إلحاق تنسيقات بالعُقد مباشرةً، أو عبر تعريف قواعد تتطابق مع عُقد بعينها، ولدينا العديد من خصائص التنسيقات مثل `color` و`display`، كما تستطيع شيفرة جافاسكربت التعديل في تنسيق العنصر مباشرةً من خلال خاصية `style`.

14.15 تدريبات

14.15.1 بناء جدول

يُبنى الجدول في لغة HTML بهيكل الوسم التالي:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
```

```
  </tr>
</table>
```

ويحتوي وسم `<table>` على وسم `<tr>` يمثل الصف الواحد، ونستطيع في كل صف وضع عناصر الخلايا سواءً كانت خلايا ترويسات `<th>` أو عادية `<td>`.

ولِّد هيكل DOM لجدول يَعُد الكائنات إذا أُعطيت مجموعة بيانات لجبال ومصفوفة من الكائنات لها الخصائص `name` و`height` و`place`، بحيث يجب أن تحتوي على عمود لكل مفتاح وصَفّ لكل كائن، إضافةً إلى صف ترويسة بعناصر `<th>` في الأعلى لتسرد أسماء الأعمدة.

اكتب ذلك بحيث تنحدر الأعمدة مباشرةً من الكائنات، من خلال أخذ أسماء الخصائص للكائن الأول في البيانات، وأضف الجدول الناتج إلى العنصر الذي يحمل سمة `id` لـ `"mountains"` كي يصبح ظاهرًا في المستند.

بمجرد أن يعمل هذا، اجعل محاذاة الخلايا التي تحتوي قيمًا عدديةً إلى اليمين من خلال ضبط خاصية `style.textAlign` لها لتكون `"right"`.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<h1>Mountains</h1>

<div id="mountains"></div>

<script>
  const MOUNTAINS = [
    {name: "Kilimanjaro", height: 5895, place: "Tanzania"},
    {name: "Everest", height: 8848, place: "Nepal"},
    {name: "Mount Fuji", height: 3776, place: "Japan"},
    {name: "Vaalserberg", height: 323, place: "Netherlands"},
    {name: "Denali", height: 6168, place: "United States"},
    {name: "Popocatepetl", height: 5465, place: "Mexico"},
    {name: "Mont Blanc", height: 4808, place: "Italy/France"}
  ];

  // ضع شيفرتك هنا
</script>
```

إرشادات الحل

استخدِم `document.createElement` لإنشاء عُقد عناصر جديدة، و`document.createTextNode` لإنشاء عُقد نصية، والتابع `appendChild` لوضع العُقد داخل عُقد أخرى.

قد تريد التكرار على أسماء المفاتيح مرةً كي تملأ الصف العلوي، ثم مرةً أخرى لكل كائن في المصفوفة لتضع بيانات الصفوف، كما يمكنك استخدام `Object.keys` للحصول على مصفوفة أسماء المفاتيح من الكائن الأول.

استخدِم `document.getElementById` أو `document.querySelector` لإيجاد العُقدة التي لها سمة `id` الصحيحة، إذا أردت إضافة الجدول إلى العُقدة الأصل المناسبة.

14.15.2 جلب العناصر بأسماء وسومها

يُعيد التابع `document.getElementsByTagName` جميع العناصر الفرعية التي لها اسم وسم معيَّن. استخدِم نسختك الخاصة منه على أساس دالة تأخذ عُقدةً وسلسلةً نصيةً -هي اسم الوسم- على أساس وسائط، وتُعيد مصفوفةً تحتوي على عُقد العناصر المنحدرة منه، والتي لها اسم الوسم المعطى.

استخدِم خاصية `nodeName` لعنصر ما كي تحصل على اسم الوسم الخاص به، لكن لاحظ أن هذا سيعيد اسم الوسم بأحرف إنجليزية من الحالة الكبيرة capital، لذا يمكنك استخدام التابعين النصيين `toLowerCase` أو `toUpperCase` لتعديل حالة تلك الحروف كما تريد.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<h1>Heading with a <span>span</span> element.</h1>
<p>A paragraph with <span>one</span>, <span>two</span>
  spans.</p>

<script>
  function byTagName(node, tagName) {
    // ضع شيفرتك هنا.
  }

  console.log(byTagName(document.body, "h1").length);
  // → 1
  console.log(byTagName(document.body, "span").length);
  // → 3
```

```
  let para = document.querySelector("p");
  console.log(byTagName(para, "span").length);
  // → 2
</script>
```

إرشادات الحل

يمكن حل هذا التدريب بسهولة باستخدام دالة تعاودية كما فعلنا في دالة `talksAbout` التي تقدم شرحها هنا.

استدع `byTagname` نفسها تعاوديًا للصفوفات الناتجة ببعضها لتكون هي الخرج، أو تستطيع إنشاء دالة داخلية تستدعي نفسها تعاوديًا ولها وصول إلى رابطة مصفوفة معرَّفة في الدالة الخارجية، بحيث يمكنها إضافة العناصر التي تجدها إليها، ولا تنسى استدعاء الدالة الداخلية من الدالة الخارجية كي تبدأ العملية.

يجب أن تتحقق الدالة التعاودية من نوع العُقدة، وما يهمنا هنا هو العُقدة التي من النوع 1 أي `Node.ELEMENT_NODE`، كما علينا في مثل تلك العُقد علينا التكرار على فروعها، وننظر في كل فرع إن كان يطابق الاستعلام في الوقت نفسه الذي نستدعيه تعاوديًا فيه للنظر في فروعه هو.

14.15.3 قبعة القطة

وسِّع مثال تحريك القطة الذي سبق كي تدور القطة على جهة مقابلة من القبعة `<img src="img/hat.png">` في القطع الناقص أو اجعل القبعة تدور حول القطة أو أي تعديل يعجبك في طريقة حركتيهما.

لتسهيل موضعة الكائنات المتعددة، من الأفضل استخدام التموضع المطلق absolute positioning، وهذا يعني أن `top` و`left` تُحسبان نسبةً إلى أعلى يسار المستند.

أضف عددًا ثابتًا من البكسلات إلى قيم الموضع كي تتجنب استخدام الإحداثيات السالبة التي ستجعل الصورة تتحرك خارج الصفحة المرئية.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<style>body { min-height: 200px }</style>
<img src="img/cat.png" id="cat" style="position: absolute">
<img src="img/hat.png" id="hat" style="position: absolute">

<script>
  let cat = document.querySelector("#cat");
```

```
  let hat = document.querySelector("#hat");

  let angle = 0;
  let lastTime = null;
  function animate(time) {
    if (lastTime != null) angle += (time - lastTime) * 0.001;
    lastTime = time;
    cat.style.top = (Math.sin(angle) * 40 + 40) + "px";
    cat.style.left = (Math.cos(angle) * 200 + 230) + "px";

    // ضع شيفرتك هنا.

   requestAnimationFrame(animate);
  }
  requestAnimationFrame(animate);
</script>
```

إرشادات الحل

تقيس الدالتان `Math.cos` و`Math.sin` الزوايا بصورة نصف دائرية أي بواحدة الراديان، فإذا كانت الدائرة تساوي 2 باي 2π كما تقدَّم، فستستطيع الحصول على الزاوية المقابلة بإضافة نصف هذه القيمة -والتي تساوي باي أو π- باستخدام `Math.PI`، وبالتالي سيسهل عليك وضع القبعة على الجهة المقابلة من القطة.

دورة تطوير واجهات المستخدم

ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

15. معالجة الأحداث

إن لك سيطرة على عقلك فقط، وليس الأحداث الخارجية، وإذا تذكرت ذلك فستجد القوة.

— ماركوس أوريليوس.

تتعامل بعض البرامج مع مدخلات المستخدِم المباشرة مثل مدخلات لوحة المفاتيح والفأرة، ومثل تلك المدخلات ليس لها هيكل منظَّم بل تكون لحظية جزءًا جزءًا، ويجب أن يتعامل البرنامج معها أثناء حدوثها.

15.1 مفهوم معالجات الأحداث Events Handlers

تخيل أن هناك واجهة لا تحوي طريقةً لمعرفة المفتاح الذي ضغطت عليه على لوحة المفاتيح، إلا بقراءة حالة المفتاح الحالية، فإذا أردت التفاعل مع ضغطات المفاتيح فسيكون عليك قراءة حالة المفتاح باستمرار كي تلتقط تغيرها قبل أن يترك إصبعك المفتاح، وسيكون من الخطير إجراء أي حسابات قد تستغرق وقتًا، إذ قد تفوتك هذه الضغطة.

إنّ هذا الأسلوب متَّبع في بعض الآلات البدائية، وأفضل منه أن نجعل العتاد أو نظام التشغيل يلاحظان ضغطات المفاتيح ويضعانها في رتل، ثم يتفقد برنامج ما هذا الرتل لينظر في الأحداث المستجِدة ويتعامل مع ما يجده هناك. ويجب أن لا يهمل هذا البرنامج قراءة الرتل، بل يجب أن يتفقّده بصورة دورية، وإلا فستلاحظ أن البرنامج الذي تتعامل معه أنت غير متجاوب، ويسمى هذا الأسلوب بالاقتراع polling، لكن يفضِّل المبرمجون تجنبه، والأفضل منهما جميعًا هو جعل النظام ينبِّه شيفرة برنامجنا كلما وقع حدث ما، وتفعل المتصفحات ذلك بالسماح لنا بتسجيل الدوال على أساس معالِجات handlers لأحداث بعينها.

```
<p>اضغط على هذا المستند لتفعيل المعالِج</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

تشير رابطة `window` إلى كائن مضمَّن built-in يوفره المتصفح، يمثل نافذة المتصفح التي تحتوي على المستند، كما يسجل استدعاء التابع `addEventListener` الخاص بها الوسيط الثاني ليُستدعَى كلما وقع الحدث الموصوف في الوسيط الأول.

15.2 الأحداث وعقد DOM

يُسجَّل كل معالِج حدثًا لمتصفح في سياق ما، فقد استدعينا `addEventListener` في المثال السابق على كائن `window` لتسجيل معالِج للنافذة كلها، ويمكن العثور على مثل هذا التابع في عناصر DOM أيضًا، وفي بعض أنواع الكائنات الأخرى.

لا تُستدعى مستمِعات الأحداث event listeners إلا عند وقوع الحدث في سياق الكائن الذي تكون مسجلة عليه.

```
<button>اضغط هنا</button>
<p>لا يوجد معالِج هنا</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

يربط هذا المثال المعالج بعقدة زر، وأيّ ضغطة على هذا الزر تشغِّل المعالج، بينما لا يحدث شيء عند الضغط على بقية المستند.

يعطي إلحاق سمة `onclick` لعقدة ما التأثير نفسه، وهذا يصلح لأغلب أنواع الأحداث، إذ تستطيع إلحاق معالج من خلال سمة يكون اسمها هو اسم الحدث مسبوقًا بـ `on`، غير أن العقدة تحتوي على سمة `onclick` واحدة فقط، لذا تستطيع تسجيل معالج واحد فقط لكل عقدة بهذه الطريقة.

يسمح التابع `addEventListener` لك بأن تضيف أي عدد من المعالجات، بحيث لا تقلق من إضافتها حتى لو كان لديك معالجات أخرى للعنصر؛ أما التابع `removeEventListener` الذي تستدعيه وسائط تشبه `addEventListener`، فإنه يحذف المعالج، انظر:

```
<button>زر الضغطة الواحدة</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

يجب أن تكون الدالة المعطاة لـ `removeEventListener` لها قيمة الدالة نفسها التي أعطيت إلى `addEventListener`، بحيث إذا أردت إلغاء تسجيل معالج ما، فعليك أن تعطي الدالة الاسم `once` في المثال كي تستطيع تمرير نفس قيمة الدالة لكلا التابعين.

15.3 كائنات الأحداث

يُمرَّر كائن الحدث إلى دوال معالجات الأحداث على أساس وسيط، ويحمل معلومات إضافية عن ذلك الحدث، فإذا أردنا معرفة أي زر قد ضُغِط عليه في الفأرة مثلًا، فإننا سنبحث في خاصية `button` لكائن الحدث.

```
<button>اضغط عليّ كيفما شئت</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Left button");
    } else if (event.button == 1) {
      console.log("Middle button");
    } else if (event.button == 2) {
      console.log("Right button");
    }
  });
</script>
```

15.4 الانتشار Propagation

تستقبل المعالجات المسجَّلة مع فروع children على العقد أحداثًا تقع في هذه الفروع أيضًا، فإذا تم النقر على زر داخل فقرة ما، فسترى معالجات الأحداث في تلك الفقرة حدث النقر أيضًا.

لكن إذا كان كل من الفقرة والزر لهما معالج، فإنّ المعالج الأكثر خصوصيةً -أي المعالج الذي على الزر مثلًا- هو الذي يعمل، ويقال هنا أن الحدث ينتشر propagate إلى الخارج outward، أي من العقدة التي حدث فيها إلى جذر المستند، وبعد أن تحصل جميع المعالجات المسجلة على عقدة ما على فرصة للاستجابة للحدث، فستحصل المعالجات المسجَّلة على النافذة بأكملها على فرصتها في الاستجابة للحدث هي أيضًا.

يستطيع معالج الحدث استدعاء التابع `stopPropagation` على كائن الحدث في أي وقت لمنع المعالجات من استقبال الحدث، وهذا مفيد إذا كان لديك مثلًا زر داخل عنصر آخر قابل للنقر. ولم ترد أن تتسبب نقرات الزر في نقرات ذلك العنصر الخارجي أيضًا.

يسجِّل المثال التالي معالجات `"mousedown"` على كل من الزر والفقرة التي حوله، فحين تنقر بالزر الأيمن سيستدعي معالج الزر الذي في الفقرة التابع `stopPropagation` الذي سيمنع المعالج الذي على الفقرة من العمل، وإذا نُقر الزر بزر آخر للفأرة فسيعمل كلا المعالجين، انظر كما يلي:

```
<p>فقرة فيها  <button>زر</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

تحتوي أغلب كائنات الأحداث على خاصية `target` التي تشير إلى العقدة التي أُنشئت فيها، ونستخدم هذه الخاصية لضمان أننا لا نعالج شيئًا انتشر من عقدة لا نريد معالجتها، كما من الممكن استخدام هذه الخاصية لإلقاء شبكة كبيرة على نوع حدث معيَّن، فإذا كانت لديك عقدة تحتوي على قائمة طويلة من الأزرار مثلًا، فقد يكون أسهل أن تسجِّل معالج نقرة منفردة على العقدة الخارجية وتجعله يستخدِم خاصية `target` ليعرف إذا كان الزر قد نُقر أم لا بدلًا من تسجيل معالجات فردية لكل الأزرار.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clicked", event.target.textContent);
    }
  });
</script>
```

15.5 الإجراءات الافتراضية

إذا نقرت على رابط ما، فستذهب إلى الهدف المرتبط بهذا الرابط؛ أما إذا نقرت على السهم المشير للأسفل، فسيهبط المتصفح بالصفحة للأسفل؛ بينما إذا نقرت بالزر الأيمن، فستحصل على القائمة المختصرة، وهكذا فإن كل حدث له إجراء افتراضي مرتبط به.

وتُستدعى معالجات الأحداث جافاسكربت قبل حدوث السلوك الافتراضي في أغلب أنواع الأحداث، فإذا لم يرد المعالج وقوع هذا السلوك الاعتيادي لأنه قد عالج الحدث بالفعل فإنه يستدعي التابع `preventDefault` على كائن الحدث.

يمكن استخدام هذا لتطبيق اختصارات لوحة المفاتيح الخاصة بك أو القائمة المختصرة، كما يمكن استخدامه ليتدخل معارضًا السلوك الذي يتوقعه المستخدِم.

انظر المثال التالي لرابط لا يذهب بالمستخدِم إلى الموقع الذي يحمله:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

لا تفعل شيئًا كهذا إلا إن كان لديك سبب مقنع، إذ سينزعج مستخدِمو صفحتك من مثل هذا السلوك المفاجئ لهم.

بعض الأحداث لا يمكن اعتراضها أبدًا في بعض المتصفحات، إذ لا يمكن معالجة اختصار لوحة المفاتيح الذي يغلق اللسان الحالي -أي ctrl+w في ويندوز أو ⌘+w في ماك- باستخدام جافاسكربت مثلًا.

15.6 أحداث المفاتيح

يطلِق المتصفح الحدث `"keydown"` في كل مرة تضغط فيها مفتاحًا من لوحة المفاتيح، وكلما رفعت يدك عن المفتاح، ستحصل على الحدث `"keyup"`.

```
<p> v تصير هذه الصفحة بنفسجية إذا ضغطت مفتاح.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

يُطلَق الحدث `"keydown"` إذا ضغطت على المفتاح وتركته أو إذا ظللت ضاغطًا عليه، حيث يُطلق في كل مرة يُكرر فيها المفتاح، وانتبه لهذا إذ أنك لو أضفت زرًا إلى DOM حين يُضغط مفتاح، ثم حذفته حين يُترك المفتاح، فقد تضيف مئات الأزرار خطأً إذا ضُغط على المفتاح ضغطًا طويلًا.

ينظر المثال في خاصية `key` لكائن الحدث ليرى عن أي مفتاح هو، إذ تحمل هذه الخاصية سلسلةً نصيةً تتوافق مع الشيء الذي يُطبع على الشاشة إذا ضُغط ذلك المفتاح، عدا بعض الحالات الخاصة التي تحمل الخاصية اسم المفتاح الذي يُضغط مثل زر الإدخال `"Enter"`.

إذا ظللت ضاغطًا على مفتاح "عالي" shift ثم ضغطت على مفتاح v مثلًا، فإن ذلك قد يتسبب في حمل الخاصية لاسم المفتاح أيضًا، وعندها تتحول `"v"` إلى `"V"`، وتتغير `"1"` إلى `"!"` إذا كان هذا ما يخرجه الضغط على shift+1 على حاسوبك.

تولِّد مفاتيح التحكم مثل shift وcontrol وalt وغيرها أحداث مفاتيح مثل المفاتيح العادية، وتستطيع معرفة إذا كانت هذه المفاتيح مضغوط عليها ضغطًا مستمرًا عند البحث عن مجموعات المفاتيح من خلال النظر إلى خصائص `shiftKey` و `ctrlKey` و `altKey` و `metaKey` لأحداث لوحة المفاتيح والفأرة.

```
<p>Press Control-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!");
    }
  });
</script>
```

تعتمد عقدة DOM حيث بدأ حدث المفتاح على العنصر الذي كان نشطًا عند الضغط على المفتاح، ولا تستطيع أغلب العقد أن تكون نشطةً إلا إذا أعطيتها سمة `tabindex` على خلاف الروابط والأزرار وحقول الاستمارات، كما سنعود لحقول الاستمارات في الفصل الثامن عشر، وإذا لم يكن ثمة شيء بعينه نشطًا، فستتصرف `document.body` على أساس عقدة هدف لأحداث المفاتيح.

لا نفضِّل استخدام أحداث المفاتيح إذا كتب المستخدِم نصًا وأردنا معرفة ما يكتبه، فبعض المنصات لا تبدأ تلك الأحداث هنا كما في حالة لوحة المفاتيح الافتراضية على هواتف الأندرويد، لكن حتى لو كانت لديك لوحة مفاتيح قديمة، فإنّ بعض أنواع النصوص المدخلة لا تتطابق مع ضغطات المفاتيح تطابقًا مباشرًا، مثل برنامج محرر أسلوب الإدخال input method editor -أو IME اختصارًا- الذي يستخدمه الأشخاص الذين لا تتناسب نصوصهم مع لوحة المفاتيح، حيث تُدمج عدة نقرات لإنشاء المحارف.

إذا أرادت العناصر التي تستطيع الكتابة فيها معرفة ما يكتبه المستخدِم كما في وسوم `<input>` و`<textarea>`، فإنها تطلق أحداث `"input"` كلما غيّر المستخدِم محتواها، ومن الأفضل قراءة المحتوى الفعلي المكتوب من الحقل النشط إذا أردنا الحصول عليه الذي سيوضحه الفصل الثامن عشر.

15.7 أحداث المؤشر

توجد حاليًا طريقتان مستخدَمتان على نطاق واسع للإشارة إلى الأشياء على الشاشة: الفأرات -بما في ذلك الأجهزة التي تعمل عملها مثل لوحات اللمس touchpads وكرات التتبع- وشاشات اللمس touchscreens، وتُنتج هاتان الطريقتان نوعين مختلفَين تمامًا من الأحداث.

15.7.1 ضغطات الفأرة

يؤدي الضغط على زر الفأرة إلى إطلاق عدد من الأحداث، ويتشابه حدثَي `"mouseup"` و`"mousedown"` مع حدثَي `"keydown"` و`"keyup"`، وتنطلق عند الضغط على الزر وتركه، كما تحدث هذه على عقد DOM الموجودة أسفل مؤشر الفأرة مباشرةً عند وقوع الحدث.

ينطلق حدث `"click"` بعد حدث `"mouseup"` على العقدة الأكثر تحديدًا التي تحتوي على كل من ضغط الزر وتحريره، فإذا ضغطت على زر الفأرة في فقرة مثلًا ثم حركت المؤشر إلى فقرة أخرى وتركت الزر، فسيقع حدث `"click"` على العنصر الذي يحتوي على هاتين الفقرتين،؛ أما في حالة حدوث نقرتين بالقرب من بعضهما، فسينطلق حدث `"dblclick"` -وهو النقرة المزدوجة- بعد حدث النقرة الثانية.

يمكنك النظر إلى الخاصيتَين `clientX` و `clientY` إذا أردت الحصول على معلومات دقيقة حول هذا المكان الذي وقع فيه حدث الفأرة، إذ تحتويان على إحداثيات الحدث -بالبكسل- نسبةً إلى الركن العلوي الأيسر من النافذة، أو `pageX` و `pageY` نسبةً إلى الركن العلوي الأيسر من المستند كله، وقد تكون هذه مختلفةً عن تلك عند تمرير النافذة.

ينفِّذ المثال التالي برنامج رسم بدائي، حيث توضع نقطة أسفل مؤشر الفأرة في كل مرة تنقر فيها على المستند. وإذا أردت إلقاء نظرة على برنامج رسم أقل بدائية انظر الفصل التاسع عشر.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

15.7.2 حركة الفأرة

ينطلق حدث "mousemove" في كل مرة يتحرك مؤشر الفأرة، ويمكن استخدام هذا الحدث لتتبع موضع المؤشر، مثل أن نحتاج إلى تنفيذ بعض المهام المتعلقة بخاصية السحب drag للمؤشر.

يوضح المثال التالي برنامجًا يعرض شريطًا ويضبط معالجات أحداث كي يتحكم السحب يمينًا ويسارًا في عرض الشريط:

```
<p>اسحب الشريط لتغيير عرضه:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let lastX; // تتبع آخر حركة مراقبة للفأرة على المحور X
  let bar = document.querySelector("div");
  bar.addEventListener("mousedown", event => {
    if (event.button == 0) {
      lastX = event.clientX;
      window.addEventListener("mousemove", moved);
      event.preventDefault(); // Prevent selection
    }
  });

  function moved(event) {
    if (event.buttons == 0) {
      window.removeEventListener("mousemove", moved);
    } else {
      let dist = event.clientX - lastX;
      let newWidth = Math.max(10, bar.offsetWidth + dist);
      bar.style.width = newWidth + "px";
      lastX = event.clientX;
    }
  }
</script>
```

لاحظ أنّ معالج "mousemove" يُسجَّل على النافذة كلها، حتى لو خرج المؤشر عن الشريط أثناء تغيير عرضه، وذلك طالما أن الزر مضغوط عليه ونكون لا زلنا نريد تعديل العرض. لكن يجب أن يتوقف تغيير الحجم فور تركنا لزر الفأرة، ولضمان ذلك فإننا نستخدم خاصية buttons -لاحظ أنها جمع وليست مفردة-، والتي

تخبرنا عن الأزرار التي نضغط عليها الآن، فإذا كانت صِفرًا، فهذا يعني أن الأزرار كلها متروكة وحرة؛ أما إذا كانت ثمة أزرار مضغوط عليها، فستكون قيمة الخاصية هي مجموع رموز هذه الأزرار، إذ يحصل الزر الأيسر على الرمز 1 والأيمن على الرمز 2، والأوسط على 4، فإذا كان الزران الأيمن والأيسر مضغوطًا عليهما معًا، فستكون قيمة `buttons` هي 3.

لاحظ أن ترتيب هذه الرموز يختلف عن الترتيب الذي تستخدمه `button`، حيث يأتي الزر الأوسط قبل الأيمن، وذلك لِما ذكرنا من قبل أنّ واجهة برمجة المتصفح تفتقر إلى التناسق.

15.7.3 أحداث اللمس

صُمِّم أسلوب المتصفح ذو الواجهة الرسومية في الأيام التي كانت فيها شاشات اللمس نادرةً جدًا في السوق، ولهذا لم توضع في الحسبان كثيرًا، لذا فقد كان على المتصفحات التي جاءت في أولى الهواتف ذات شاشات اللمس التظاهر بأن أحداث اللمس هي نفسها أحداث الفأرة -وإن كان إلى حد ما-، فإذا نقرت على شاشتك فستحصل على الأحداث `"mousedown"` و `"mouseup"` و `"click"`.

لكن هذا المنظور ركيك بما أنّ شاشة اللمس تعمل بأسلوب مختلف تمامًا عن الفأرة، فلا توجد هنا أزرار متعددة ولا يمكن تتبع الإصبع إذا لم يكن على الشاشة فعلًا لمحاكاة `"mousemove"`، كما تسمح الشاشة بعدة أصابع عليها في الوقت نفسه.

لا تغطي أحداث الفأرة شاشات اللمس إلا في حالات مباشرة، فإذا أضفت معالج `"click"` إلى زر ما، فسيستطيع المستخدِم الذي يستعمل شاشة لمس استخدام الزر هنا، لكن لن يعمل مثال الشريط السابق على شاشة لمس.

كما أن هناك أنواعًا بعينها من الأحداث تنطلق عند التفاعل باللمس فقط، فحين يلمس الإصبع الشاشة، فستحصل على حدث `"touchstart"`، وإذا تحرك أثناء اللمس فستُطلَق أحداث `"touchmove"`؛ أما إذا ابتعد عن الشاشة فستحصل على حدث `"touchend"`.

تمتلك كائنات هذه الأحداث خاصية `touches` التي تحمل كائنًا شبيهًا بالمصفوفة من نقاط لكل منها خصائص `cientX` و `clientY` و `pageX` و `pageY`، وذلك لأنّ كثيرًا من شاشات اللمس تدعم اللمس المتعدد في الوقت نفسه، فلا يكون لتلك الأحداث مجموعةً واحدةً فقط من الأحداث.

تستطيع فعل شيء مشابه لتظهر دوائر حمراء حول كل إصبع يلمس الشاشة:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
```

```
<p>Touch this page</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>
```

قد ترغب في استدعاء `preventDefault` في معالجات أحداث اللمس لتنسخ -أي تُعدِّل- سلوك المتصفح الافتراضي الذي قد يشمل تمرير الشاشة عند تحريك الإصبع للأعلى أو الأسفل لتمرير الصفحة، ولمنع أحداث المؤشر من الانطلاق، والتي سيكون لديك معالج لها أيضًا.

15.8 أحداث التمرير

ينطلق حدث `"scroll"` كلما مُرِّر عنصر ما، ونستطيع استخدام ذلك في معرفة ما الذي ينظر إليه المستخدِم الآن كي نوقف عمليات التحريك أو الرسوم المتحركة التي خرجت من النطاق المرئي للشاشة -أو لإرسال هذه البيانات إلى جامعي بيانات المستخدِمين من مخترقي الخصوصية-، أو لإظهار تلميح لمدى تقدم المستخدِم في الصفحة بتظليل عنوان في الفهرس أو إظهار رقم الصفحة أو غير ذلك.

يرسم المثال التالي شريط تقدم أعلى المستند ويحدِّثه ليمتلئ كلما مررت للأسفل:

```
<style>
  #progress {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
```

```
      top: 0; left: 0;
    }
  </style>
  <div id="progress"></div>
  <script>
    // اكتب محتوى هنا
    document.body.appendChild(document.createTextNode(
      "supercalifragilisticexpialidocious ".repeat(1000)));

    let bar = document.querySelector("#progress");
    window.addEventListener("scroll", () => {
      let max = document.body.scrollHeight - innerHeight;
      bar.style.width = `${(pageYOffset / max) * 100}%`;
    });
  </script>
```

إذا كان الموضع `position` الخاص بالعنصر ثابتًا `fixed`، فسيتصرف كما لو كان له موضع مطلق `absolute`، لكنه يمنعه من التمرير مع بقية المستند، ويُترجَم هذا التأثير في الحالات الواقعية على أساس حالة شريط التقدم في مثالنا، إلا أننا نريد جعل الشريط ظاهرًا في أعلى الصفحة أو المستند بغض النظر عن موضع التمرير فيه، ويتغير عرضه ليوضِّح مدى تقدمنا في المستند، كما سنستخدم `%` بدلًا من `px` لضبط وحدة العرض كي يكون حجم العنصر نسبيًا لعرض الصفحة.

تعطينا الرابطة العامة `innerheight` ارتفاع النافذة التي يجب طرحها من الارتفاع الكلي الذي يمكن تمريره، بحيث لا يمكن التمرير بعد الوصول إلى نهاية المستند، ولدينا بالمثل `innerwidth` للعرض الخاص بالنافذة؛ وتحصل على النسبة الخاصة بشريط التمرير عبر قسمة موضع التمرير الحالي `pageYoffset` على أقصى موضع تمرير وتضرب الناتج في 100.

كذلك لا يُستدعى معالج الحدث إلا بعد وقوع التمرير نفسه، وبالتالي لن يمنع استدعاء `preventDefault` وقوع حدث التمرير.

15.9 أحداث التنشيط Focus Events

إذا كان عنصر ما نشطًا، فسيطلق المتصفح حدث `"focus"` عليه، وإذا فقد نشاطه ذلك بانتقال التركيز منه إلى غيره فإنه يحصل على حدث `"blur"`. لا ينتشر هذان الحدثان على عكس الأحداث السابقة، ولا يُبلَّغ المعالج الذي على العنصر الأصل حين يكون عنصر فرعي نشطًا أو حين يفقد نشاطه.

يوضح المثال التالي نص مساعدة لحقل نصي نشط:

```
<p>الاسم: <input type="text" data-help="اسمك الكامل"></p>
<p>العمر: <input type="text" data-help="عمرك بالأعوام"></p>
<p id="help"></p>

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
</script>
```

سيستقبل كائن النافذة حدثي `"focus"` و`"blur"` كلما تحرك المستخدِم من وإلى نافذة المتصفح أو اللسان النشط الذي يكون المستند معروضًا فيه.

15.10 حدث التحميل Load Event

ينطلق حدث `"load"` على النافذة وكائنات متن المستند إذا أتمت صفحة ما تحميلها، وهو يُستخدَم عادةً لجدولة إجراءات التهيئة initialization actions التي تكون في حاجة إلى بناء المستند كاملًا.

تذكَّر أنّ محتوى وسوم `<script>` يُشغَّل تلقائيًا إذا قابل الوسم، وقد يكون هذا قبل أوانه إذا احتاجت السكربت إلى فعل شيء بأجزاء المستند التي تظهر بعد وسم `<script>` مثلًا.

تمتلك العناصر التي تحمِّل ملفًا خارجيًا -مثل الصور ووسوم السكربت- حدث `"load"` كذلك، حيث يوضِّح تحميل الملفات التي تشير إليها، وهذه الأحداث -أي أحداث التحميل- لا تنتشر propagate أي مثل الأحداث المتعلقة بالنشاط focus.

حين نغلق صفحةً ما أو نذهب بعيدًا عنها إلى غيرها عند فتح رابط مثلًا، فسينطلق حدث `"beforeunload"`، والاستخدام الرئيسي لهذا الحدث هو منع المستخدِم من فقد أي عمل كان يعمله إذا أُغلق المستند، فإذا منعت السلوك الافتراضي لهذا الحدث وضبطت خاصية `returnvalue` على كائن

الحدث لتكون سلسلة نصية؛ فسيظهر المتصفح للمستخدِم صندوقًا حواريًا يسأله إذا كان يرغب في ترك الصفحة حقًا.

قد يحتوي الصندوق الحواري سلسلتك النصية التي تحاول الحفاظ على بيانات المستخدِم فعليًا، لكن كثيرًا من المواقع كانت تستخدِم هذا الأسلوب من أجل وضع المستخدِمين في حيرة وخداعهم ليبقوا على صفحات هذه المواقع ويشاهدوا الإعلانات الموجودة هناك، لكن المتصفحات لم تَعُد تظهر هذه الرسائل في الغالب.

15.11 الأحداث وحلقات الأحداث التكرارية

تتصرف معالجات أحداث المتصفح في سياق حلقة الحدث التكرارية مثل إشعارات غير متزامنة كما ناقشنا في الفصل الحادي عشر، وتُجدوَل حين يقع الحدث، لكن عليها الانتظار حتى تنتهي السكربتات العاملة أولًا قبل أن تعمل هي.

يعني هذا أنه إذا كانت حلقة الحدث التكرارية مرتبطةً بمهمة أخرى، فإن أي تفاعل مع الصفحة -وهو ما يحدث أثناء الأحداث- سيتأخر حتى نجد وقتًا لمعالجته، لذلك إذا كانت لديك مهام كثيرة مجدولة إما مع معالج حدث يستغرق وقتًا طويلًا أو مع معالجات لأحداث قصيرة لكنها كثيرة جدًا، فستصير الصفحة بطيئةً ومزعجةً في الاستخدام.

أما إذا أردت فعل شيء يستغرق وقتًا في الخلفية دون التأثير على أداء الصفحة، فستوفِّر المتصفحات شيئًا اسمه عمّال الويب web workers، ويُعَدّ ذاك العامل في جافاسكربت مهمةً تعمل إلى جانب السكربت الرئيسية على الخط الزمني الخاص بها.

تخيَّل أنّ تربيع عدد ما يمثل عمليةً حسابيةً طويلةً وثقيلة، وأننا نريد إجراءها في خيط thread منفصل، فنكتب حينها ملفًا اسمه `code/squareworker.js` يستجيب للرسائل بحساب التربيع وإرساله في رسالة.

```
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```

لا تشارك العمال نطاقها العام أو أي بيانات أخرى مع البيئة الرئيسية للسكربت لتجنب مشاكل الخيوط المتعددة التي تتعامل مع البيانات نفسها، وعليك التواصل معها عبر إرسال الرسائل ذهابًا وعودة.

ينتج المثال التالي عاملًا يشغِّل تلك السكربت ويرسل بعض الرسائل إليها ثم يخرج استجاباتها:

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("The worker responded:", event.data);
});
```

```
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

ترسِل دالة `postMessage` رسالةً تطلق حدث `"message"` في المستقبِل، كما ترسل السكربت التي أنشأت العامل رسائلًا، وتستقبلها من خلال كائن `Worker`؛ في حين يخاطب العامل السكربت التي أنشأته عبر الإرسال مباشرةً على نطاقها العام والاستماع إليه.

يمكن للقيم التي تمثل على أساس JSON أن تُرسَل على أساس رسائل، وسيَستقبل الطرف المقابل نسخةً منها بدلًا من القيمة نفسها.

15.12 المؤقتات Timers

رأينا دالة `setTimeout` في الفصل الحادي عشر وكيف أنها تجدوِل دالةً أخرى لتُستدعى لاحقًا بعد وقت محدد يُحسب بالميلي ثانية، لكن أحيانًا قد تحتاج إلى إلغاء دالة جدولتها بنفسك سابقًا، ويتم هذا بتخزين القيمة التي أعادتها `setTimeout` واستدعاء `clearTimeout` عليها.

```
let bombTimer = setTimeout(() => {
  console.log("بووم!");
}, 500);
if (Math.random() < 0.5) { // احتمال 50%
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

تعمل الدالة `cancelAnimationFrame` بالطريقة نفسها التي تعمل بها `clearTimeout`، أي أنّ استدعاءها على قيمة أعادتها `requestAnimationFrame`، سيلغي هذا الإطار، وذلك على افتراض أنه لم يُستدعى بعد، كما تُستخدَم مجموعة مشابهة من الدوال هي `setInterval` و`clearInterval` لضبط المؤقتات التي يجب أن تتكرر كل عدد معيّن X من الميللي ثانية.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

15.13 تقييد إطلاق الحدث Debouncing

بعض أنواع الأحداث لها قابلية الانطلاق بسرعة وعدة مرات متتابعة مثل حدثي "mousemove" و"scroll"، وحين نعالج هذه الأحداث، يجب الحذر من فعل أيّ شيء يستغرق وقتًا كبيرًا وإلا فسيأخذ معالجنا وقتًا طويلًا بحيث يبطئ التفاعل مع المستند.

إذا أردت فعل شيء مهم بهذا المعالج، فيمكنك استخدام setTimeout للتأكد أنك لا تفعله كثيرًا، ويسمى هذا بتقييد إطلاق الحدث event debouncing.

إذا كتب المستخدم شيئًا ما فإننا نريد التفاعل معه في المثال الأول هنا، لكن لا نريد فعل ذلك فور كل حدث إدخال، فإذا كان يكتب بسرعة، فسنريد الانتظار حتى يتوقف ولو لبرهة قصيرة، كما نضبط مهلةً بدلًا من تنفيذ إجراء مباشرةً على معالج الحدث، إلى جانب حذفنا لأي مهلة زمنية timeout سابقة أقرب من تأخير مهلتنا الزمنية، كما ستُلغى المهلة الزمنية التي من الحدث السابق.

```
<textarea>اكتب شيئًا هنا...</textarea>
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Typed!"), 500);
  });
</script>
```

إنّ إعطاء قيمة غير معرَّفة لـ clearTimeout أو استدعاءها على مهلة زمنية أُطلِقت سلفًا، ليس له أي تأثير، وعليه فلا داعي لأن تخشى شيئًا إذا أردت استدعاءها، بل افعل ذلك لكل حدث إذا شئت.

تستطيع استخدام نمط pattern مختلف قليلًا إذا أردت المباعدة بين الاستجابات بحيث تكون مفصولةً بأقل مدة زمنية محددة، لكن في الوقت نفسه تريد إطلاقها أثناء سلسلة أحداث -وليس بعدها-، حيث يمكنك مثلًا الاستجابة إلى أحداث "mousemove" بعرض الإحداثيات الحالية للفأرة، لكن تعرضها كل 250 مللي ثانية، انظر منا يلي:

```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
```

```
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
</script>
```

15.14 خاتمة

تمكننا معالجات الأحداث من استشعار الأحداث التي تحدث في صفحة الويب والتفاعل معها، ويُستخدَم التابع `addEventListener` لتسجيل مثل تلك المعالِجات.

كل حدث له نوع يعرِّفه مثل `"keydown"` و `"focus"` وغيرهما، وتُستدعى أغلب الأحداث على عنصر DOM بعينه، ثم ينتشر إلى أسلاف هذا العنصر سامحًا للمعالجات المرتبطة بتلك العناصر أن تعالجها.

عندما يُستدعى معالج حدث ما، فسيُمرَّر إليه كائن حدث بمعلومات إضافية عن الحدث، وذلك الكائن له تابع يسمح لنا بإيقاف الانتشار `stopPropagation`، وآخر يمنع معالجة المتصفح الافتراضية للحدث `preventDefault`.

إذا ضغطنا مفتاحًا فسيطلق هذا حدثَي `"keydown"` و `"keyup"`؛ أما الضغط على زر الفأرة فسيطلق الأحداث `"mousedown"` و `"mouseup"` و `"click"` في حين يطلق تحريك المؤشر أحداث `"mousemove"` كما يطلق التفاعل مع شاشات اللمس الأحداث `"touchstart"` و`"touchmove"` و `"touchend"`.

يمكن استشعار التمرير scrolling من خلال حدث `"scroll"`، كما يمكن استشعار تغيرات النافذة محل التركيز أو النافذة النشطة من خلال حدثَي `"focus"` و `"blur"`، وإذا أنهى المستند تحميله، فسينطلق حدث `"load"` للنافذة.

15.15 تدريبات

15.15.1 بالون

اكتب صفحةً تعرض بالونًا باستخدام الصورة الرمزية للبالون balloon emoji، بحيث يكبر هذا البالون بنسبة 10% إذا ضغطت السهم المشير للأعلى، ويصغر إذا ضغطت على السهم المشير للأسفل بنسبة 10%.

تستطيع التحكم في حجم النص -بما أن الصورة الرمزية ما هي إلا نص- بضبط `font-size` لخاصية `style.fontSize` على العنصر الأصل لها، وتذكر ألا تنسى ذكر وحدة القياس في القيمة مثل كتابة `10px`.

تأكد من أن المفاتيح تغير البالون فقط دون تمرير الصفحة، وأن أسماء مفاتيح الأسهم هي `"ArrowUp"` و `"ArrowDown"`. وإذا نجح ذلك فأضف ميزةً أخرى هي انفجار البالون عند بلوغه حجمًا معينًا، ويعني هذا هنا استبدال الصورة الرمزية للانفجار بالصورة الرمزية للبالون، ويُزال معالج الحدث هنا كي لا تستطيع تغيير حجم الانفجار. تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<p>🎈</p>

<script>
  // شيفرتك هنا
</script>
```

إرشادات الحل

يجب تسجيل معالج لحدث `"keydown"` وأن تنظر في `event.key` لتعرف هل ضُغِط السهم الأعلى أم الأسفل، ويمكن الحفاظ على الحجم الحالي في رابطة binding كي تستطيع بناء الحجم الجديد عليه، وسيكون هذا نافعًا -سواءً الرابطة، أو نمط البالون في الـ DOM- في تعريف الدالة التي تحدث الحجم، وذلك كي تستدعيها من معالج الحدث الخاص بك، وربما كذلك بمجرد البدء لضبط الحجم الابتدائي.

تستطيع تغيير البالون إلى انفجار عبر استبدال عقدة النص بأخرى باستخدام `replaceChild`، أو بضبط خاصية `textContent` لعقدتها الأصل على سلسلة نصية جديدة.

15.15.2 ذيل الفأرة

كان يعجب الناس في الأيام الأولى لجافاسكربت أن تكون صفحات المواقع ملأى بالصور المتحركة المبهرجة والتأثيرات البراقة، وأحد هذه التأثيرات هو إعطاء ذيل لمؤشر الفأرة في صورة سلسلة من العناصر التي تتبع المؤشر في حركته في الصفحة، وفي هذا التدريب نريدك تنفيذ ذيل للمؤشر.

استخدم عناصر `<div>` التي لها مواضع مطلقة بحجم ثابت ولون خلفية -حيث يمكنك النظر في فقرة ضغطات الفأرة لتكون مرجعًا لك-، وأنشئ مجموعةً من هذه العناصر واعرضها عند تحرك المؤشر لتكون في عقبه مباشرةً.

لديك عدة طرق تحل بها هذا التدريب، والأمر إليك إذا شئت جعل الحل سهلًا أو صعبًا؛ فالحل السهل هو أن تجعل عددًا ثابتًا من العناصر وتجعلها في دورة لتحرك العنصر التالي إلى الموضع الحالي للفأرة في كل مرة يقع حدث `"mousemove"`.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<style>
  .trail { /* className for the trail elements */
    position: absolute;
    height: 6px; width: 6px;
    border-radius: 3px;
    background: teal;
  }
  body {
    height: 300px;
  }
</style>

<script>
  // ضع شيفرتك هنا.
</script>
```

إرشادات الحل

يفضَّل إنشاء العناصر باستخدام حلقة تكرارية، وتُلحق العناصر بالمستند كي تظهر، كما ستحتاج إلى تخزين هذه العناصر في مصفوفة كي تستطيع الوصول إليها لاحقًا لتغيير موقعها.

يمكن تنفيذ الدورة عليها بتوفير متغير عدّاد وإضافة 1 إليه في كل مرة ينطلق فيها حدث `"mousemove"`، بعدها يمكن استخدام عامل `% elements.length` للحصول على فهرس مصفوفة صالحة لاختيار العنصر الذي تريد موضعته خلال الحدث الذي لديك.

تستطيع أيضًا تحقيق تأثير جميل عبر نمذجة نظام فيزيائي بسيط باستخدام حدث `"mousemove"` لتحديث زوج من الرابطات التي تتبع موضع الفأرة، ثم استخدام `requestAnimationFrame` لمحاكاة العناصر اللاحقة التي تنجذب إلى موضع مؤشر الفأرة. حدِّث موضعها في كل خطوة تحريك وفقًا لموضعها النسبي للمؤشر وربما سرعتها أيضًا إذا خزنتها لكل عنصر، وسنترك لك التفكير في طريقة جيدة لفعل ذلك.

15.15.3 التبويبات Tabs

تُستخدم اللوحات المبوَّبة في واجهات المستخدِم بكثرة، إذ تسمح لك باختيار لوحة من خلال اختيار التبويب الذي في رأسها، وفي هذا التدريب ستنفذ واجهةً مبوبةً بسيطةً.

اكتب الدالة `asTabs` التي تأخذ عقدة DOM وتنشئ واجهةً مبوبةً تعرض العناصر الفرعية من تلك العقدة، ويجب إدخال قائمة من عناصر `<buttons>` في أعلى العقدة، بحيث يكون عندك واحد لكل عنصر فرعي، كما يحتوي على نص يأتي من سمة `data-tabname` للفرع.

يجب أن تكون كل العناصر الفرعية الأصلية مخفيةً عدا واحدًا منها -أي تعطيها قيمة none لنمط display- كما يمكن اختيار العقدة المرئية الآن عبر النقر على الأزرار.

وسِّع ذلك إذا نجح معك لتنشئ نمطًا لزر التبويب المختار يختلف عما حوله ليُعلم أي تبويب تم اختياره.

```
<tab-panel>
  <div data-tabname="one">التبويب الأول</div>
  <div data-tabname="two">التبويب الثاني</div>
  <div data-tabname="three">التبويب الثالث</div>
</tab-panel>
<script>
  function asTabs(node) {
    // ضع شيفرتك هنا.
  }
  asTabs(document.querySelector("tab-panel"));
</script>
```

إرشادات الحل

إحدى المشاكل التي قد تواجهها هي أنك لن تستطيع استخدام خاصية childNodes الخاصة بالعقدة استخدامًا مباشرًا مثل تجميعة لعقد التبويب، ذلك لأنك حين تضيف الأزرار، إذ أنها ستصبح عقدًا فرعيةً كذلك وتنتهي في ذلك الكائن لأنه هيكل بيانات حي، كذلك فإنّ العقد النصية المنشأة للمسافات الفارغة بين العقد هي عناصر فرعية childNodes أيضًا، ويجب ألا تحصل على تبويبات خاصة بها، وبالتالي استخدم children بدلًا من childNodes لتجاهل العقد النصية.

قد تبدأ ببناء مصفوفة تبويبات كي يكون لديك وصول سهل لها، وتستطيع تخزين الكائنات التي تحتوي كلًا من لوحة التبويب والزر الخاص بها لتنفيذ التنسيق styling الخاص بالأزرار، كما ننصحك بكتابة دالة منفصلة لتغيير التبويبات؛ فإما تخزين التبويب المختار سابقًا وتغيير الأنماط المطلوب إخفاؤها وعرض الجديدة فقط، أو تحديث نمط جميع التبويبات في كل مرة يُختار تبويب جديد فيها.

قد تريد استدعاء هذه الدالة فورًا لتبدأ الواجهة مع أول تبويب مرئي.

دورة تطوير التطبيقات
باستخدام لغة JavaScript

احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

16. مشروع لعبة منصة

ما الواقع إلا لعبة كبيرة.

— إيان بانكس Iain Banks، لاعب الألعاب The Player of Games

بدأ ولع الكثير من العاملين في مجالات التقنية بالحواسيب منذ الصغر من خلال ألعاب الحاسوب، خاصةً تلك التي فيها عوالم افتراضية يستطيع اللاعب فيها التحكم في سير اللعبة وأحداث القصة التي فيها، ورغم تشابه مجال برمجة الألعاب مع مجال الموسيقى من حيث نسبة العرض إلى الطلب، إلا أن التناقض العجيب بين كمية المبرمجين الصغار الذي يرغبون في العمل فيها ومقدار الطلب الحقيقي على أولئك المبرمجين يخلق بيئة عمل في غاية السوء، لكن بأي حال فذاك لا يمنع أنّ كتابة الألعاب قد تكون مسليةً للبعض.

سنتعلم هنا كيفية كتابة لعبة منصة صغيرة، وتُعَدّ لعب المنصات platform games -أو ألعاب "اقفز واركض"- ألعابًا تتوقع من اللاعب تحريك شخصية في عالم افتراضي يكون ثنائي الأبعاد غالبًا، كما يكون منظور اللعبة من الجانب مع القفز على عناصر وكائنات أو القفز خلالها.

16.1 اللعبة

ستبنى لعبتنا بصورة ما على Dark Blue التي كتبها توماس بالِف Thomas Palef، وقد اخترنا هذه اللعبة لأنها مسلية وصغيرة الحجم في نفس الوقت، كما يمكن بناؤها دون كتابة الكثير من الشيفرات وستبدو في النهاية هكذا:

يمثِّل اللاعب بالصندوق الداكن الذي تكون مهمته جمع الصناديق الصفراء -أي العملات النقدية- مع تجنب الكائنات الحمراء -أي الحمم البركانية-، ويكتمل المستوى حين تُجمع كل العملات النقدية.

يستطيع اللاعب التحرك في اللعبة باستخدام أسهم اليمين واليسار، كما يستطيع القفز باستخدام زر السهم الأعلى، ويكون للقفز هنا ميزةً ليست واقعيةً لكنها تعطي اللاعب إحساس التحكم بالشخصية التي على الشاشة، وهي أن الشخصية تستطيع القفز لمسافة أكبر بعدة مرات من طولها، كما تستطيع تغيير اتجاهها في الهواء.

تتكون اللعبة من خلفية ثابتة توضع في هيئة شبكة خلفية، وتكون العناصر المتحركة فوق تلك الخلفية، كما يكون كل حقل في تلك الشبكة إما خاليًا أو مصمتًا أو يحوي حممًا؛ أما العناصر المتحركة فتكون اللاعب أو العملات أو أجزاء من الحمم، كما لا تتقيد مواضع تلك العناصر بالشبكة بل قد تكون إحداثياتها أعدادًا كسْرية لتسمح لها بالحركة بنعومة.

16.2 التقنية

سنستخدم نموذج كائن المستند DOM الخاص بالمتصفح لعرض اللعبة، وسنقرأ مدخلات المستخدِم من خلال معالجة أحداث المفاتيح.

تمثِّل الشيفرة المتعلقة بالشاشة ولوحة المفاتيح جزءًا صغيرًا من العمل الذي علينا تنفيذه لبناء هذه اللعبة، ولن يكون رسم اللعبة صعبًا بما أنها تتكون من صناديق ملونة في الغالب، حيث سننشئ عناصر DOM وسنستخدم التنسيقات لنعطيها لون خلفية وحجمًا وموضعًا أيضًا.

تمثَّل الخلفية على أساس جدول بما أنها شبكة مربعات ثابتة؛ أما العناصر التي تتحرك بحرية فتمثَّل باستخدام عناصر مطلقة الموضع absolutely positioned، وبما أننا نريد تمثيل مدخلات اللاعب والتجاوب معها دون تأخير ملحوظ فستكون الكفاءة مهمة هنا، ورغم أن DOM لم يُصمم للرسوميات عالية الأداء، إلا أن أداءه أفضل مما هو متوقع، فقد رأينا بعض ذلك في الفصل الرابع عشر، وسيكون أداء مثل هذه اللعبة ممتازًا على حاسوب حديث حتى لو لم نحسِّن أداءها كثيرًا.

لكن مع هذا فسننظر في تقنية أخرى للمتصفحات في الفصل التالي، وهي وسم <canvas> الذي يوفر طريقةً تقليديةً أكثر لتصميم الرسوميات، إذ يتعامل مع الأشكال والبكسلات بدلًا من عناصر DOM.

16.3 المستويات

نريد طريقةً لتحديد مستويات اللعبة بحيث يسهل للمستخدِم قراءتها وتعديلها أيضًا، وبما أن كل شيء هنا يمكن إنشاؤه على شبكة، فسنستخدم سلاسل نصية طويلة يمثِّل كل محرف فيها عنصرًا، بحيث يكون إما جزءًا من شبكة الخلفية أو عنصرًا متحركًا، كما سيبدو السطح الذي يمثل أحد المستويات الصغيرة كما يلي:

```
let simpleLevelPlan = `
......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#++++++++++++#..
......##############..
......................`;
```

تمثِّل النقاط المساحات الفارغة، وتمثِّل محارف الشباك # الحوائط؛ أما علامات الجمع فتمثِّل الحمم البركانية، ويكون موضع بدء اللاعب عند العلامة @، كما يمثِّل كل محرف o في المستوى هنا عملة نقدية، وتمثل علامة = كتلةً من الحمم تتحرك جيئة وذهابًا بصورة أفقية.

سنضيف نوعين آخرين من الحمم المتحركة، حيث سيمثِّل الأول محرف الأنبوب | للحمم المتحركة رأسيًا، ومحرف v للحمم المتساقطة، وهي حمم متحركة رأسيًا لا تتردد بين نقطتين، وإنما تتحرك للأسفل فقط قافزةً إلى نقطة بدايتها حين تصل إلى القاع.

تتكون اللعبة كلها من مستويات عدة يجب على اللاعب إكمالها، ويكتمل المستوى حين تُجمع كل العملات كما ذكرنا؛ أما إذا لمس اللاعب حممًا بركانيةً فسيعود المستوى الحالي إلى نقطة البداية ليحاول اللاعب مرةً أخرى.

16.4 قراءة المستوى

يخزن الصنف التالي كائن المستوى، وسيكون وسيطه هو السلسلة النصية التي تعرِّف المستوى.

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l]);
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];

    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
        let type = levelChars[ch];
        if (typeof type == "string") return type;
        this.startActors.push(
          type.create(new Vec(x, y), ch));
        return "empty";
      });
    });
  }
}
```

يُستخدَم التابع `trim` لحذف المسافات الفارغة في بداية ونهاية السلسلة النصية لسطح المستوى level plan، وهذا يسمح للسطح في مثالنا أن يبدأ بسطر جديد كي تكون جميع الأسطر تحت بعضها مباشرةً، ثم تقسَّم السلسلة الباقية بمحارف أسطر جديدة، وينتشر كل سطر في مصفوفة لتكون عندنا مصفوفات من المحارف، وبناءً عليه تحمل `rows` مصفوفةً من مصفوفات المحارف تكون هي صفوف سطح المستوى، كما نستطيع أخذ عرض المستوى وطوله منها، لكن لا زال علينا فصل العناصر المتحركة من شبكة الخلفية.

سنسمي العناصر المتحركة باسم الكائنات الفاعلة أو actors، والتي ستخزَّن في مصفوفة من الكائنات؛ أما الخلفية فستكون مصفوفةً من مصفوفات سلاسل نصية تحمل أنواعًا من الحقول مثل `"empty"` أو `"wall"` أو `"lava"`.

سنمر على الصفوف ثم على محتوياتها من أجل إنشاء تلك المصفوفات، وتذكَّر أنّ `map` تمرِّر فهرس المصفوفة على أنه الوسيط الثاني إلى دالة الربط mapping function التي تخبرنا إحداثيات x وy لأي عنصر، كما ستخزَّن المواضع في اللعبة على أساس أزواج من الإحداثيات بحيث يكون الزوج الأعلى إلى اليسار هو 0,0، ثم يكون عرض وارتفاع كل مربع في الخلفية هو وحدة واحدة.

يستخدِم الباني `level` كائن `levelChars` لاعتراض الكائنات في سطح المستوى، وهو يربط عناصر الخلفية بالسلاسل، ويربط المحارف الفاعلة actor characters بالأصناف. وحين يكون `type` صنفَ كائن

فاعل actor، فسيُستخدم التابع الساكن `create` الخاص به لإنشاء كائن يضاف إلى `startActors`، ثم تعيد دالة الربط `"empty"` لمربع الخلفية ذاك.

يخزِّن موضع الكائن الفاعل على أساس كائن `Vec` الذي هو متجه ثنائي الأبعاد، أي كائن له خصائص `x` و `y` كما رأينا في الفصل السادس.

ستتغير مواضع الكائنات الفاعلة مع تشغيل اللعبة لتكون في أماكن مختلفة أو حتى تختفي تمامًا كما في حال العملات عند جمعها، ولهذا سنستخدم الصنف `State` لتتبّع حالة اللعبة أثناء عملها.

```
class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }

  static start(level) {
    return new State(level, level.startActors, "playing");
  }

  get player() {
    return this.actors.find(a => a.type == "player");
  }
}
```

ستتغير الخاصية `status` لتكون `"lost"` أو `"won"` عند نهاية اللعبة، ونكون هنا مرةً أخرى أمام هيكل بيانات ثابت، إذ ينشئ تحديث حالة اللعبة حالةً جديدةً ويترك القديمة كما هي.

16.5 الكائنات الفاعلة Actors

تمثل الكائنات الفاعلة الموضع والحالة الحاليَين لعنصر معطى في اللعبة، وتعمل كلها بالواجهة نفسها، كما تحمل الخاصية `pos` الخاصة بها إحداثيات الركن العلوي الأيسر للعنصر، في حين تحمل خاصية `size` حجمه.

ثم إن لديها التابع `update` الذي يُستخدَم لحساب حالتها الجديدة وموضعها كذلك بعد خطوة زمنية معطاة، ويحاكي الإجراء الذي يأخذه الكائن الفاعل -الاستجابة لأزرار الأسهم والتحرك وفقها بالنسبة للاعب، والقفز للأمام أو الخلف بالنسبة للحمم-، ويعيد كائن فاعل `actor` جديدًا ومحدَّثًا.

تحتوي الخاصية `type` على سلسلة نصية تعرف نوع الكائن الفاعل سواءً كان `"player"` أو `"coin"` أو `"lava"`، وهذا مفيد عند رسم اللعبة، حيث سيعتمد مظهر المستطيل المرسوم من أجل كائن فاعل على نوعه.

تحتوي أصناف الكائنات الفاعلة على التابع الساكن `create` الذي يستخدمه الباني `Level` لإنشاء كائن فاعل من شخصية في مستوى السطح، كما يعطى إحداثيات الشخصية والشخصية نفسها، إذ هي مطلوبة لأن صنف `Lava` يعالج عدة شخصيات مختلفة.

لدينا فيما يلي الصنف `Vec` الذي سنستخدمه من أجل قيمنا ثنائية البعد مثل موضع الكائنات الفاعلة وحجمها.

```
class Vec {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vec(this.x + other.x, this.y + other.y);
  }
  times(factor) {
    return new Vec(this.x * factor, this.y * factor);
  }
}
```

يغيِّر التابع `times` حجم المتجه بعدد معيّن، والذي سيفيدنا حين نحتاج إلى زيادة متجه السرعة بضربه في مدة زمنية لنحصل على المسافة المقطوعة خلال تلك المدة.

تحصل الأنواع المختلفة من الكائنات الفاعلة على أصنافها الخاصة بما أنّ سلوكها مختلف. دعنا نعرِّف تلك الأصناف وسننظر لاحقًا في توابع `update` الخاصة بها. سيكون لصنف اللاعب الخاصية `speed` التي تخزن السرعة الحالية لتحاكي قوة الدفع والجاذبية.

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }
```

```
  static create(pos) {
    return new Player(pos.plus(new Vec(0, -0.5)),
                      new Vec(0, 0));
  }
}

Player.prototype.size = new Vec(0.8, 1.5);
```

يكون الموضع الابتدائي للاعب فوق الموضع الذي يظهر فيه محرف @ بنصف مربع، وذلك لأن طول اللاعب يساوي مربعًا ونصف، وتكون بهذه الطريقة قاعدته بمحاذاة قاعدة المربع الذي يظهر فيه.

كذلك تكون الخاصية `size` هي نفسها لجميع نُسخ `Player`، لذا نخزنها في النموذج الأولي بدلًا من النُسخ نفسها، وقد كان بإمكاننا استخدام جالبة مثل `type`، ولكنه سينشئ ويعيد ذلك كائن `Vec` جديد في كل مرة تُقرأ فيها الخاصية، وهو هدر لا حاجة له، إذ لا نحتاج إلى إعادة إنشاء السلاسل النصية في كل مرة نقيِّمها بما أنها غير قابلة للتغير immutable.

عند إنشاء الكائن الفاعل `Lava` سنحتاج إلى تهيئته initialization تهيئةً مختلفةً وفقًا للمحرف المبني عليه، فالحمم الديناميكية تتحرك بسرعتها الحالية إلى أن تصطدم بعائق، فإذا كانت لديها الخاصية `reset` فستقفز إلى موضع البداية -أي تقطير dripping-؛ أما إذا لم تكن لديها، فستعكس سرعتها وتكمل في الاتجاه المعاكس -أي ارتداد bouncing-.

ينظر التابع `create` في المحرف الذي يمرره الباني `Level` وينشئ كائن الحمم الفاعل المناسب.

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
```

```
        return new Lava(pos, new Vec(0, 2));
      } else if (ch == "v") {
        return new Lava(pos, new Vec(0, 3), pos);
      }
    }
  }

  Lava.prototype.size = new Vec(1, 1);
```

أما كائنات `Coin` الفاعلة فهي بسيطة نسبيًا، إذ تظل في مكانها لا تتحرك، لكن سنعطيها تأثيرًا متمايلًا بحيث تتحرك رأسيًا جيئةً وذهابًا، كما يخزن كائن العملة موضعًا أساسيًا وخاصية `wobble` تتتبَّع مرحلة حركة الارتداد من أجل تتبعها، ويحددان معًا الموضع الحقيقي للعملة ويُخزَّن في الخاصية `pos`.

```
  class Coin {
    constructor(pos, basePos, wobble) {
      this.pos = pos;
      this.basePos = basePos;
      this.wobble = wobble;
    }

    get type() { return "coin"; }

    static create(pos) {
      let basePos = pos.plus(new Vec(0.2, 0.1));
      return new Coin(basePos, basePos,
                      Math.random() * Math.PI * 2);
    }
  }

  Coin.prototype.size = new Vec(0.6, 0.6);
```

رأينا في الفصل الرابع عشر أنّ `Math.sin` تعطينا إحداثية y لنقطة ما في دائرة، وتتذبذب تلك الإحداثية في حركة موجية ناعمة أثناء الحركة على الدائرة، مما يعطينا دالةً جيبيةً نستفيد منها في نمذجة الحركة الموجية.

سنجعل مرحلة بدء كل عملة عشوائية، وذلك لكي نتفادى صنع حركة متزامنة للعملات، ويكون عرض الموجة التي تنتجها `Math.sin` هو 2π وهي مدة الموجة، ثم نضرب القيمة المعادة بـ `Math.random` بذلك العدد لتعطي العملة موضع بدء عشوائي على الموجة.

نستطيع الآن تعريف كائن `levelChars` الذي يربط محارف السطح لأنواع شبكة الخلفية أو لأصناف الكائنات الفاعلة.

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

يعطينا ذلك جميع الأجزاء التي نحتاج إليها لإنشاء نسخة `Level`.

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9
```

ستكون المهمة التالية هي عرض تلك المستويات على الشاشة وضبط الوقت والحركة فيها.

16.6 مشكلة التغليف

لا يؤثر التغليف encapsulation على أغلب الشيفرات الموجودة في هذا الفصل لسببين، أولهما أنه يأخذ جهدًا إضافيًا فيجعل البرامج أكبر ويتطلب مفاهيم وواجهات إضافية، وبما أن تلك شيفرات كثيرة على القارئ، فقد اجتهدنا في تصغير البرنامج وتبسيطه؛ أما الثاني أن ثمة عناصر عديدة في اللعبة ترتبط ببعضها، بحيث إذا تغير سلوك أحدها، فمن الصعب بقاء غيرها كما هو، وسيجعل ذلك الواجهات تطرح الكثير من الافتراضات عن طريقة عمل اللعبة، مما يقلل من فائدتها، فإذا غيرنا جزءًا من النظام، فسيكون علينا مراقبة تأثير ذلك في الأجزاء الأخرى بما أنّ واجهاتها لن تغطي الموقف الجديد.

بعض النقاط الفاصلة تقبل الانفصال من خلال واجهات صارمة، لكن هذا ليس حال كل النقاط لدينا، كما سنهدر طاقةً كبيرةً في محاولة تغليف شيء لا يكون حدًا مناسبًا، وسنلاحظ إذا ارتكبنا ذلك الخطأ أنّ واجهاتنا صارت كبيرةً وكثيرة التفاصيل، كما ستحتاج إلى التغيير كل حين كلما تغير البرنامج.

لكن ثمة شيء سنغلفه هو النظام الفرعي للرسم، وذلك لأننا سنعرض اللعبة نفسها بطريقة مختلفة في الفصل التالي، فإذا وضعنا الرسم خلف واجهة، فسنتمكن من تحميل برنامج اللعبة نفسه، ونلحقه بوحدة العرض الجديدة.

16.7 الرسم

تُغلَّف شيفرة الرسم من خلال تعريف كائن عرض display object يعرض مستوى ما وحالته، وسيكون اسم نوع العرض الذي نعرّفه في هذا الفصل هو `DOMDisplay` بما أنه يستخدم عناصر DOM لعرض المستوى، كما

سنستخدم ورقة أنماط style sheet لضبط الألوان الفعلية والخصائص الثابتة الأخرى للعناصر التي ستشكِّل اللعبة، كذلك يمكن تعيين الخاصية `style` للعناصر مباشرةً حين ننشئها لكن سينتج ذلك برامج أكثر إسهابًا.

توفر الدالة المساعدة التالية طريقةً موجزةً لإنشاء عنصر وتعطيه بعض السمات والعقد الفرعية:

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

يُنشأ العرض بإعطائه كائن مستوى وعنصرًا أبًا parent element ليلحِق نفسه به.

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

تُرسم شبكة الخلفية للمستوى مرةً واحدةً بما أنها لن تتغير، ويعاد رسم الكائنات الفاعلة في كل مرة يُحدّث فيها العرض بحالة ما، كما سنستخدم الخاصية `actorLayer` لتتبّع العنصر الذي يحمل الكائنات الفاعلة، بحيث يمكن حذفها واستبدالها بسهولة؛ كما ستُتتبّع إحداثياتنا وأحجامنا بوحدات الشبكة، حيث يشير الحجم أو المسافة التي تساوي 1 إلى كتلة شبكة واحدة، ويجب أن نزيد حجم الإحداثيات حين نضبط أحجام البكسلات، إذ سيبدو كل شيء صغيرًا في اللعبة إذا جعلنا كل بكسل يقابل مربعًا واحدًا، وسيعطينا الثابت `scale` عدد البكسلات التي تأخذها واحدةً واحدة single unit على الشاشة.

```
const scale = 20;

function drawGrid(level) {
```

```
    return elt("table", {
      class: "background",
      style: `width: ${level.width * scale}px`
    }, ...level.rows.map(row =>
      elt("tr", {style: `height: ${scale}px`},
          ...row.map(type => elt("td", {class: type})))
    ));
  }
```

تُرسم الخلفية على أساس عنصر `<table>`، ويتوافق ذلك بسلاسة مع هيكل الخاصية `rows` للمستوى، فقد حُول كل صف في الشبكة إلى صف جدول -أي عنصر `<tr>`-؛ أما السلاسل النصية في الشبكة فتُستخدم على أساس أسماء أصناف لخلية الجدول -أي `<td>`- كما يُستخدم عامل النشر spread operator -أي النقطة الثلاثية- لتمرير مصفوفات من العقد الفرعية إلى `elt` لفصل الوسائط.

يوضح المثال التالي كيف نجعل الجدول يبدو كما نريد من خلال شيفرة CSS:

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

تُستخدم بعض الوسوم مثل `table-layout` و`border-spacing` و`padding`، لمنع السلوك الافتراضي غير المرغوب فيه، فلا نريد لتخطيط الجدول أن يعتمد على محتويات خلاياه، كما لا نريد مسافات بين خلايا الجدول أو تبطينًا padding داخلها.

تضبط قاعدة `background` لون الخلفية، إذ تسمح CSS بتحديد الألوان على أساس كلمات -مثل `white`- أو بصيغة مثل `rgb(R, G, B)`، حيث تُفصل مكونات اللون الحمراء والخضراء والزرقاء إلى ثلاثة أعداد من 0 إلى 255. ففي اللون `rgb(52, 166, 251)` مثلًا، سيكون مقدار المكون الأحمر 52 والأخضر 166 والأزرق 251، وبما أن مقدار الأزرق هو الأكبر، فسيكون اللون الناتج مائلًا للزرقة، ويمكن رؤية ذلك في القاعدة `.lava`، إذ أنّ أول عدد فيها -أي الأحمر- هو الأكبر.

سنرسم كل كائن فاعل بإنشاء عنصر DOM له وضبط موضع وحجم ذلك العنصر بناءً على خصائص الكائن الفاعل، ويجب ضرب القيم في `scale` لتُحوَّل من وحدات اللعبة إلى بكسلات.

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}
```

تُفصل أسماء الأصناف بمسافات كي نعطي العنصر الواحد أكثر من صنف، ففي شيفرة CSS أدناه سنرى أن الصنف `actor` يعطي الكائنات الفاعلة موضعها المطلق، كما يُستخدم اسم نوعها على أساس صنف إضافي ليعطيها لون، ولا نريد تعريف صنف `lava` مرةً أخرى بما أننا سنعيد استخدامه من أجل مربعات شبكة الحمم التي عرّفناها سابقًا.

```
.actor  { position: absolute;            }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

يُستخدَم التابع `syncState` لجعل العرض يظهر حالةً ما، وهو يحذف رسوميات الكائن الفاعل القديم أولًا إذا وُجدت، ثم يعيد رسم الكائنات الفاعلة في مواضعها الجديدة.

قد يكون من المغري استخدام عناصر DOM للكائنات الفاعلة، لكننا سنحتاج إلى الكثير من الحسابات الإضافية إذا أردنا إنجاح ذلك من أجل ربطها مع عناصر DOM، ولضمان حذف العناصر حين تختفي كائناتها الفاعلة، وعلى أيّ حال فليست لدينا كائنات فاعلة كثيرة في اللعبة، وبالتالي لن تكون إعادة رسمها مكلفةً.

```
DOMDisplay.prototype.syncState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

نستطيع تخصيص الكائن الفاعل للاعب تخصيصًا مختلفًا حين تفوز اللعبة أو تخسر بإضافة حالة المستوى الحالية، مثل اسم صنف إلى المغلِّف، وذلك من خلال إضافة قاعدة CSS لا تأخذ ذلك التأثير إلا عندما يكون للاعب عنصر سلف بصنف ما.

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

يتغير لون اللاعب إلى الأحمر الداكن بعد لمس الحمم ليشير إلى الحرق، كما نضيف إليه هالةً بيضاء حوله إذا جمع كل العملات، وذلك بإضافة ظلَّين أبيضين ضبابيين، بحيث يكون واحدًا أعلى يساره والثاني أعلى يمينه.

لا يمكن افتراض تلاؤم المستوى مع نافذة الرؤية على الدوام، ونافذة الرؤية هي العنصر الذي سنرسم اللعبة داخله، لذا نحتاج إلى استدعاء `scrollPlayerIntoView` الذي يضمن أننا سنمرر scroll نافذة الرؤية إذا كان المستوى سيخرج عنها إلى أن يصير اللاعب قريبًا من مركزها.

تعطي شيفرة CSS التالية قيمة حجم عظمى لعنصر DOM المغلِّف الخاص باللعبة، ويضمن ألا يُرى شيء خارج صندوق العنصر، كما سنعطيه موضعًا نسبيًا كي تكون مواضع الكائنات الفاعلة داخلها منسوبةً إلى الركن العلوي الأيسر من المستوى.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

نبحث عن موضع اللاعب في التابع `scrollPlayerIntoView` ونحدِّث موضع التمرير للعنصر المغلِّف، كما نغير موضع التمرير بتعديل الخصائص `scrollLeft` و`scrollTop` الخاصة بالعنصر حين يقترب اللاعب من الحافة.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;
```

```
    // The viewport
    let left = this.dom.scrollLeft, right = left + width;
    let top = this.dom.scrollTop, bottom = top + height;

    let player = state.player;
    let center = player.pos.plus(player.size.times(0.5))
                           .times(scale);

    if (center.x < left + margin) {
      this.dom.scrollLeft = center.x - margin;
    } else if (center.x > right - margin) {
      this.dom.scrollLeft = center.x + margin - width;
    }
    if (center.y < top + margin) {
      this.dom.scrollTop = center.y - margin;
    } else if (center.y > bottom - margin) {
      this.dom.scrollTop = center.y + margin - height;
    }
  };
```

تُظهِر الطريقة التي نحدد بها مركز اللاعب كيفية سماح التوابع التي على نوع `Vec` بكتابة حسابات الكائنات بطريقة قابلة للقراءة نوعًا ما، ونفعل ذلك بإضافة موضع الكائن الفاعل -وهنا ركنه العلوي الأيسر- ونصف حجمه، ويكون هذا هو المركز في إحداثيات المستوى، لكننا سنحتاج إليه في إحداثيات البكسل أيضًا، لذا نضرب المتجه الناتج بمقياس العرض.

تبدأ بعد ذلك سلسلة من التحقُّقات للتأكد من أن موضع اللاعب داخل المجال المسموح به، وقد يؤدي ذلك أحيانًا إلى تعيين إحداثيات تمرير غير منطقية، كأن تكون قيمًا سالبة أو أكبر من مساحة العنصر القابلة للتمرير، ولا بأس في هذا، إذ ستقيد عناصر DOM تلك القيم لتكون في نطاق مسموح به، فإذا ضُبطت `srollLeft` لتكون `-10` مثلًا، فإنها ستتغير لتصير `0`.

قد يقال أن الأسهل يكون بجعل اللاعب يُمرَّر دائمًا ليكون في منتصف الشاشة، غير أنّ هذا -وإذا كان أسهل- سيُحدِث أثرًا عكسيًا من حيث تجربة استخدام للعبة، فكلما قفز اللاعب، ستتغير نافذة الرؤية للأعلى والأسفل معه، ولهذا من الأفضل حينها إبقاء منطقة محايدة ثابتة في منتصف الشاشة كي نتحرك دون التسبب في أي تمرير.

نستطيع الآن عرض المستوى الصغير الذي أنشأناه.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```

يُستخدَم الوسم `<link>` مع `rel="stylesheet"` لتحميل ملف CSS إلى الصفحة، ويحتوي ملف `game.css` على الأنماط الضرورية للعبتنا.

16.8 الحركة والتصادم

نحن الآن في مرحلة نستطيع فيها إضافة الحركة، وهي لا شك بأنها أكثر جزء مثير في اللعبة، والطريقة التي تتبعها أغلب الألعاب التي تشبه لعبتنا هي تقسيم الوقت إلى خطوات صغيرة ونحرك الكائنات الفاعلة بمسافة تتوافق مع سرعتها ومضروبة في حجم الخطوة الزمنية، كما سنحسب الزمن بالثواني، وعليه سيعبَّر عن السرعات بوحدات لكل ثانية.

يُعَدّ تحريك الكائنات أمرًا يسيرًا؛ أما التعامل مع التفاعلات التي يجب حدوثها بين العناصر فهو الأمر الصعب، فإذا اصطدم اللاعب بجدار أو أرضية، فلا يجب المرور من خلالها، بل تنتبه اللعبة إذا تسببت حركة ما في ارتطام كائن بآخر، ثم تتصرف وفق الحالة، فتتوقف الحركة مثلًا بالنسبة للجدران؛ أما إذا اصطدم بعملة ما فيجب جمعها، وإذا لمس حممًا بركانية فيجب خسارة اللعبة.

لا شك أن هذه عملية معقدة إذا أردنا ضبط قوانينها، لذا ستجد مكتبات يُطلق عليها عادةً محركات فيزيائية physics engines تحاكي التفاعل بين الكائنات الفيزيائية في بعدين أو ثلاثة أبعاد، ولكن بأي حال سنلقي نظرةً عليها في هذا الفصل لنتعرف إليها، حيث سنعالج التصادمات بين الكائنات المستطيلة فقط على أساس تدريب عملي عليها.

ننظر أولًا قبل تحريك اللاعب أو كتلة الحمم هل تأخذه الحركة داخل جدار أم لا، فإذا أخذته؛ فإننا نلغي الحركة بالكلية. تعتمد الاستجابة لمثل ذلك التصادم على نوع الكائن الفاعل نفسه، فاللاعب مثلًا سيقف، بينما ترتد كتلة الحمم في الاتجاه المعاكس.

يتطلب هذا الأسلوب أن تكون خطواتنا الزمنية صغيرةً إلى حد ما بما أنها ستوقف الحركة قبل أن تتلامس الكائنات، وإلا سنجد اللاعب يحوم لمسافة ملحوظة فوق الأرض إذا كانت الخطوات الزمنية كبيرةً -وخطوات الحركة بناءً عليها-، والأفضل هنا هو أن نجد نقطة التصادم بالتحديد وننتقل إليها، لكن هذا أعقد، لذا سنأخذ الأسلوب البسيط ونتلافى مشاكله بالتأكد من أن الحركة مستمرة في خطوات صغيرة.

يخبرنا التابع التالي هل يلمس المستطيل -المحدَّد بموضع وحجم- عنصر شبكة من نوع ما أم لا.

```
Level.prototype.touches = function(pos, size, type) {
  let xStart = Math.floor(pos.x);
  let xEnd = Math.ceil(pos.x + size.x);
  let yStart = Math.floor(pos.y);
  let yEnd = Math.ceil(pos.y + size.y);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
                      y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

يحسب التابع مجموعة مربعات الشبكة التي يتداخل الجسم معها من خلال استخدام `Math.floor` و`Math.ceil` على إحداثياته، وتذكَّر أنّ مربعات الشبكة حجمها 1 × 1 وحدة، فإذا قربنا جوانب الصندوق لأعلى وأسفل سنحصل على مجال مربعات الخلفية التي يلمسها الصندوق.

سنمر حلقيًا على كتلة مربعات الشبكة التي خرجنا بها من الإحداثيات السابقة، ونعيد القيمة `true` إذا وجدنا مربعًا مطابِقًا. تُعامَل المربعات التي خارج المستوى على أساس جدار `"wall"` لضمان عدم خروج اللاعب من العالم الافتراضي، وأننا لن نقرأ أي شيء خارج حدود مصفوفتنا `rows` بالخطأ.

تابع الحالة `update` يستخدِم `touches` لمعرفة هل لمس اللاعب حممًا بركانية أم لا.

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);
```

```
    if (newState.status != "playing") return newState;

    let player = newState.player;
    if (this.level.touches(player.pos, player.size, "lava")) {
      return new State(this.level, actors, "lost");
    }

    for (let actor of actors) {
      if (actor != player && overlap(actor, player)) {
        newState = actor.collide(newState);
      }
    }
    return newState;
  };
```

تُمرَّر كل من الخطوة الزمنية وهيكل البيانات إلى التابع، حيث يخبرنا هيكل البيانات بالمفاتيح المضغوط عليها حاليًا، ويستدعي التابعَ `update` على جميع الكائنات الفاعلة ليُنتج لنا مصفوفةً من نسخها المحدثة، كما تحصل الكائنات الفاعلة على الخطوة الزمنية والمفاتيح والحالة أيضًا لتتمكن من بناء تحديثها عليها، لكن تلك المفاتيح لا يقرؤها إلا اللاعب نفسه بما أنه هو الكائن الفاعل الوحيد الذي تتحكم به لوحة المفاتيح.

إذا انتهت اللعبة، فلا يكون قد بقي شيء من المعالجة لفعله، أي أن اللعبة يستحيل أن تفوز بعد خسارتها أو العكس؛ أما إذا لم تنتهي، فسيختبر التابع هل لمس اللاعب حمم الخلفية أم لا، فإذا لمسها تخسر اللعبة وتنتهي.

أخيرًا، إذا كانت اللعبة لا تزال قائمةً فسيرى هل تداخلت كائنات فاعلة أخرى مع اللاعب أم لا، ويُكتشف التداخل بين الكائنات الفاعلة باستخدام الدالة `overlap` التي تأخذ كائنين فاعلين وتعيد true إذا تلامسا فقط، وهي الحالة التي يتداخلا فيها على محوري x وy معًا.

```
  function overlap(actor1, actor2) {
    return actor1.pos.x + actor1.size.x > actor2.pos.x &&
           actor1.pos.x < actor2.pos.x + actor2.size.x &&
           actor1.pos.y + actor1.size.y > actor2.pos.y &&
           actor1.pos.y < actor2.pos.y + actor2.size.y;
  }
```

فإذا تداخل كائن فاعل، فسيحصل التابع `collide` الخاص به على فرصة لتحديث حالته، ويضبط لمس كائن الحمم حالة اللعبة إلى `"lost"`؛ أما العملات فستختفي حين نلمسها، وعند لمس العملة الأخيرة تُضبَط حالة اللعبة إلى `"won"`.

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

16.9 تحديثات الكائنات الفاعلة

تأخذ توابع `update` الخاصة بالكائنات الفاعلة الخطوة الزمنية وكائن الحالة وكائن `keys` على أساس وسائط لها، مع استثناء تابع الكائن الفاعل `Lava`، إذ يتجاهل كائن `keys`.

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

يحسب التابع `update` موضعًا جديدًا بإضافة ناتج الخطوة الزمنية والسرعة الحالية إلى الموضع القديم، فإذا لم تحجب ذلك الموضع الجديد أية عوائق فسينتقل إليه؛ أما إذا وُجد عائق فسيعتمد السلوك حينها على نوع كتلة الحمم، فالحمم المتساقطة لديها الموضع `reset` الذي تقفز إليه حين تصطدم بشيء ما؛ أما الحمم المرتدة، فتعكس سرعتها وتضربها في `-1` كي تبدأ بالحركة في الاتجاه المعاكس.

تستخدِم العملات كذلك التابع `update` من أجل تأثير التمايل، فتتجاهل التصادمات مع الشبكة بما أنها تتمايل في المربع نفسه.

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vec(0, wobblePos)),
                  this.basePos, wobble);
};
```

تتزايد الخاصية `wobble` لتراقب الوقت، ثم تُستخدَم على أساس وسيط لـ `Math.sin` لإيجاد الموضع الجديد على الموجة، ثم يُحسب موضع العملة الحالي من موضعها الأساسي وإزاحة مبنية على هذه الموجة.

يتبقى لنا اللاعب نفسه، إذ تُعالَج حركة اللاعب معالجةً مستقلةً لكل محور، ذلك أنه ينبغي على اصطدامه بالأرض ألا يمثِّل مشكلةً وألا يمنع الحركة الأفقية، كما أن الاصطدام بحائط لا يجب ألا يوقف حركة السقوط أو القفز.

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vec(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vec(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall")) {
    pos = movedY;
```

```
    } else if (keys.ArrowUp && ySpeed > 0) {
      ySpeed = -jumpSpeed;
    } else {
      ySpeed = 0;
    }
    return new Player(pos, new Vec(xSpeed, ySpeed));
  };
```

تُحسب الحركة الأفقية وفقًا لحالة مفاتيح الأسهم اليمين واليسار، فإذا لم يكن هناك حائط يحجب الموضع الجديد الذي سينشأ بسبب تلك الحركة، فسيُستخدَم؛ وإلا نظل على الموضع القديم.

كذلك الأمر بالنسبة للحركة الرأسية، لكن يجب محاكاة القفز والجاذبية، فتتزايد سرعة اللاعب الرأسية `ySpeed` أولًا لتعادل الجاذبية (تراجع)، ثم نتحقق من الحوائط مرةً أخرى، فإذا لم نصطدم بحائط فسنستخدم الموضع الجديد؛ أما إذا اصطدمنا بحائط فلدينا احتمالان، وهما إما أن يُضغط زر السهم الأعلى ونحن نتحرك إلى الأسفل، أي أن ما اصطدمنا به كان تحتنا، فتُضبط السرعة على قيمة كبيرة سالبة، وهذا يجعل اللاعب يقفز لأعلى، ويُعَد ما سوى ذلك اصطدام اللاعب بشيء في طريقه، وهنا تتغير السرعة إلى صفر؛ أما قوة الجاذبية وسرعة القفز وغيرها من الثوابت في تلك اللعبة، فتُضبط بالتجربة والخطأ، حيث اختبرنا القيم حتى وصلنا إلى قيم متوافقة مع بعضها بعضًا ومناسبة.

16.10 مفاتيح التعقب

لا نريد للمفاتيح أن تُحدِث تأثيرًا واحدًا لكل نقرة عليها، بل نريد أن يظل تأثيرها عاملًا طالما كان المفتاح مضغوطًا، وذلك مفيد في شأن لعبة مثل التي نكتبها من أجل إجراء ما مثل تحريك اللاعب.

سنُعِدّ معالِج مفتاح يخزن الحالة الراهنة لمفاتيح الأسهم الأربعة، كما سنستدعي `preventDefault` لتلك المفاتيح كي لا تتسبب في تمرير الصفحة.

تعيد الدالة في المثال أدناه كائنًا عند إعطائها مصفوفةً من أسماء المفاتيح، حيث يتعقب الموضع الحالي لتلك المفاتيح ويسجل معالجات أحداث للأحداث `"keydown"` و`"keyup"`، وإذا كانت شيفرة المفتاح التي في الحدث موجودةً في مجموعة الشيفرات التي تتعقبها، فستحدِّث الكائن.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
```

```
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

تُستخدَم الدالة المعالج نفسها لنوعي الأحداث، إذ تنظر في الخاصية `type` لكائن الحدث لتحدِّد هل يجب تحديث حالة المفتاح إلى القيمة true -أي `"keydown"`- أو القيمة false -أي `"keyup"`-.

16.11 تشغيل اللعبة

توفِّر الدالة `requestAnimationFrame` التي رأيناها في الفصل الرابع عشر طريقةً جيدةً لتحريك اللعبة، لكن واجهتها بدائية للغاية، إذ نحتاج إلى تعقب الوقت الذي استدعيت فيه دالتنا في آخر مرة، ثم نستدعي `requestAnimationFrame` مرةً أخرى بعد كل إطار ولهذا سنعرِّف دالةً مساعدةً تغلّف هذه الأجزاء المملة في واجهة مريحة وعملية، وتسمح لنا باستدعاء `runAnimation` ببساطة لتعطيها دالةً تتوقع فرق الوقت على أساس وسيط وترسم إطارًا واحدًا، كما تتوقف الحركة إذا أعادت دالة الإطار القيمة `false`.

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

ضبطنا القيمة العظمى لخطوة الإطار لتكون مساويةً لـ 100 ميلي ثانية -أي عُشر ثانية-، فإذا أُخفيت نافذة المتصفح أو التبويب الذي فيه صفحتنا، فستتوقف استدعاءات `requestAnimationFrame` إلى أن يُعرَض

التبويب أو النافذة مرةً أخرى، ويكون الفرق في هذه الحالة بين `lastTime` و`time` هو الوقت الكلي الذي أُخفيت فيه الصفحة.

لا شك في أن تقدُّم اللعبة بذلك المقدار في خطوة واحدة سيكون سخيفًا وقد يسبب آثارًا جانبيةً غريبةً، كأن يسقط اللاعب داخل الأرضية.

تحوِّل الدالة كذلك الخطوات الزمنية إلى ثواني، وهي أسهل في النظر إليها على أساس كمية عنها إذا كانت بالمللي ثانية، وتأخذ الدالة `runLevel` الكائن `Level` وتعرض بانيًا وتُعيد وعدًا، كما تعرض المستوى -في `document.body`-، وتسمح للمستخدِم باللعب من خلاله، وإذا انتهى المستوى بالفوز أو الخسارة، فستنتظر `runLevel` زمنًا قدره ثانيةً واحدةً إضافيةً ليتمكن المستخدِم من رؤية ما حدث، ثم تمحو العرض وتوقف التحريك، وتحل الوعد لحالة اللعبة النهائية.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

تتكون اللعبة من سلسلة من المستويات، بحيث يعاد المستوى الحالي إذا مات اللاعب، وإذا اكتمل مستوى، فسننتقل إلى المستوى التالي، ويمكن التعبير عن ذلك بالدالة التالية التي تأخذ مصفوفةً من أسطح المستويات (سلاسل نصية) وتعرض بانيًا:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

لأننا جعلنا `runLevel` تعيد وعدًا، فيمكن كتابة `runGame` باستخدام دالة `async` كما هو موضح في الحادي عشر، وهي تُعيد وعدًا آخرًا يُحَل عندما يُنهي اللاعب اللعبة.

ستجد مجموعةً من أسطح المستويات متاحةً في رابطة `GAME_LEVELS` في صندوق الاختبارات الخاص بهذا الفصل، وتغذي تلك الصفحة المستويات إلى `runGame` لتبدأ اللعبة الحقيقية.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

جرب بنفسك لترى ما إذا كنت تستطيع تجاوز هذه المستويات.

16.12 تدريبات

16.12.1 انتهاء اللعبة

من المتعارف عليه في ألعاب الحاسوب أنّ اللاعب يبدأ بعدد محدود من فرص الحياة التي تنقص بمقدار حياة واحدة كلما مات في اللعبة، وإذا انتهت الفرص المتاحة، فستعيد اللعبة التشغيل من البداية.

عدّل `runGame` لتضع فيها خاصية الحيوات، واجعل اللاعب يبدأ بثلاثة حيوات، ثم أخرج عدد الحيوات الحالي باستخدام `console.log` في كل مرة يبدأ فيها مستوى.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
   // ... القديمة، عدّلها runGame دالة
  async function runGame(plans, Display) {
    for (let level = 0; level < plans.length;) {
      let status = await runLevel(new Level(plans[level]),
                                  Display);
      if (status == "won") level++;
    }
    console.log("You've won!");
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

16.12.2 الإيقاف المؤقت للعبة

أضف خاصية الإيقاف المؤقت للعبة والعودة إليها من خلال مفتاح Esc، ويمكن تنفيذ هذا بتغيير دالة `runLevel` لتستخدِم معالِج حدث لوحة مفاتيح آخر، وتعترض الحركة أو تستعيدها كلما ضغط اللاعب على زر Esc.

قد لا تبدو واجهة `runAnimation` مناسبةً لهذه الخاصية، لكنها ستكون كذلك إذا أعدت ترتيب الطريقة التي تستدعيها `runLevel` بها.

إذا تمكنت من تنفيذ ذلك فثمة شيء آخر قد تستطيع فعله، ذلك أنّ الطريقة التي نسجل بها معالِجات الأحداث تسبب لنا مشكلة، فالكائن `arrowKeys` حاليًا هو رابطة عامة global binding، وتظل معالجات أحداثه باقيةً حتى لو لم تكن هناك لعبة تعمل، فتستطيع القول أنها تتسرب من نظامنا. وسّع `trackKeys` من أجل توفير طريقة لتسجيل معالجاتها عندما تبدأ ثم تلغي تسجيلها مرةً أخرى عند انتهائها.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // The old runLevel function. Modify this...
  function runLevel(level, Display) {
    let display = new Display(document.body, level);
    let state = State.start(level);
    let ending = 1;
    return new Promise(resolve => {
      runAnimation(time => {
        state = state.update(time, arrowKeys);
        display.syncState(state);
        if (state.status == "playing") {
          return true;
        } else if (ending > 0) {
          ending -= time;
          return true;
        } else {
          display.clear();
          resolve(state.status);
          return false;
        }
      });
    });
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

إرشادات الحل

يمكن اعتراض الحركة بإعادة `false` من الدالة المعطاة لـ `runAnimation`، ويمكن متابعتها باستدعاء `runAnimation` مرةً أخرى، وهكذا سنحتاج إلى إبلاغ الدالة المعطاة لـ `runAnimation` أننا سنوقف اللعبة مؤقتًا؛ ولفعل هذا، استخدِم رابطةً يستطيع كل من معالج الحدث والدالة الوصول إليها.

عند البحث عن طريقة لإلغاء تسجيل المعالجات المسجَّلة بواسطة `trackKeys`، تذكر أنّ قيمة الدالة الممررة نفسها إلى `addEventListener` يجب تمريرها إلى `removeEventListener` من أجل حذف معالج بنجاح، وعليه يجب أن تكون قيمة الدالة `handler` المنشأة في `trackKeys` متاحةً في الشيفرة التي تلغي تسجيل المعالِجات. تستطيع إضافة خاصية إلى الكائن المعاد بواسطة `trackKeys` تحتوي على قيمة الدالة أو على تابع يعالج إلغاء التسجيل مباشرةً.

16.12.3 الوحش

من الشائع أيضًا في ألعاب المنصة أن تحتوي على أعداء تستطيع القفز فوقها لتتغلب عليها، ويطلب منك هذا التدريب إضافة مثل نوع الكائن الفاعل ذلك إلى اللعبة.

سنطلق عليه اسم الوحش وتتحرك تلك الوحوش أفقيًا فقط، كما تستطيع جعلها تتحرك في اتجاه اللاعب وتقفز للأمام والخلف مثل الحمم الأفقية، أو يكون لها أي نمط حركة تختاره، ولا تحتاج إلى جعل الصنف يعالج السقوط، لكن يجب التأكد من أن الوحش لا يسير خلال الجدران.

يتوقف التأثير الواقع على اللاعب إذا لمسه أحد الوحوش بكون اللاعب يقفز فوق الوحش أم لا، ويمكنك تقريب الأمر بالتحقق من قاعدة اللاعب هل هي قريبة من قمة الوحش أم لا، فإذا كانت قريبةً فسيختفي الوحش، وإذا كانت بعيدةً فستخسر اللعبة.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<link rel="stylesheet" href="css/game.css">
<style>.monster { background: purple }</style>

<body>
  <script>
    // أكمل التوابع التالية: constructor وupdate وcollide
    class Monster {
      constructor(pos, /* ... */) {}

      get type() { return "monster"; }

      static create(pos) {
        return new Monster(pos.plus(new Vec(0, -1)));
      }
```

```
        update(time, state) {}

        collide(state) {}
      }

      Monster.prototype.size = new Vec(1.2, 2);

      levelChars["M"] = Monster;

      runLevel(new Level(`
    ..................................
    .################################.
    .#..............................#.
    .#..............................#.
    .#..............................#.
    .#...........................o..#.
    .#..@........................#.
    .##########.............########.
    ..........#..o..o..o..o..#........
    ..........#...........M..#........
    ..........################........
    ..................................
    `), DOMDisplay);
      </script>
    </body>
```

إرشادات الحل

إذا أردت تنفيذ نوع حركة حالي stateful مثل الارتداد، فتأكد من تخزين الحالة المطلوبة في الكائن الفاعل، بأن تضمّنها على أساس وسيط باني وتضيفها على أساس خاصية.

تذكر أنّ `update` تعيد كائنًا جديدًا بدلًا من تغيير الكائن القديم، وابحث عن اللاعب في `state.actors` عند معالجة اصطدام ووازن موضعه مع موضع الوحش.

للحصول على قاعدة اللاعب يجب عليك إضافة حجمه الرأسي إلى موضعه الرأسي، وسيمثل إنشاء حالة محدثة إما التابع `collide` الخاص بـ `Coin`، وهو ما يعني حذف الكائن الفاعل، أو ذلك الخاص بـ `Lava`، والذي سيغير الحالة إلى `"lost"` وفقًا لموضع اللاعب.

بعيد

هل تريد كتابة سيرة ذاتية احترافية؟

نساعدك في إنشاء سيرة ذاتية احترافية عبر خبراء توظيف
مختصين في أكبر منصة توظيف عربية عن بعد

أنشئ سيرتك الذاتية الآن

17. الرسم على لوحة

الرسم خدعة.

— إم سي إسكر M.C. Escher، مقتبس بواسطة برونو إرنست Bruno Ernst في المرآة السحرية لإم سي إسكر.

تعطينا المتصفحات طرقًا عدةً لعرض الرسوميات على الشاشة، وأبسط تلك الطرق هي استخدام التنسيقات لموضعة وتلوين عناصر شجرة DOM العادية، ويمكن فعل الكثير بهذا كما رأينا في اللعبة التي في الفصل السابق، كما يمكننا جعل العقد كما نريد بالضبط من خلال إضافة صور خلفية شبه شفافة إليها، أو أن نديرها أو نزخرفها باستخدام التنسيق `transform`، لكننا سنستخدِم عناصر DOM هكذا لغير الغرض الذي صُممت له، كما ستكون بعض المهام مثل رسم سطر بين نقطتين عشوائيتين غريبةً إذا نفذناها باستخدام عناصر HTML عادية.

لدينا بديلين هنا، حيث أن البديل الأول مبني على DOM ويستخدِم الرسوميات المتجهية القابلة للتحجيم Scalable Vector Graphics -أو SVG اختصارًا- بدلًا من HTML، كما يمكن النظر إلى SVG على أنها صيغة توصيف مستندات تركِّز على الأشكال بدلًا من النصوص، ويمكن تضمين مستند SVG مباشرةً في مستند HTML أو إدراجه باستخدام الوسم `<img>`؛ أما البديل الثاني فيدعى اللوحة Canvas، وهو عنصر DOM واحد يغلف صورةً ما، ويوفر واجهةً برمجيةً لرسم الأشكال على مساحة تشغلها عقدة ما.

الفرق الأساسي بين اللوحة وصورة SVG هو أن الوصف الأصلي للشكل في الأخيرة محفوظ بحيث يمكن نقله أو إعادة تحجيمه في أيّ وقت؛ أما اللوحة فتحوِّل الأشكال إلى بكسلات -وهي النقاط الملونة على الشاشة- بمجرد رسمها، كما لا تتذكر ما تمثله تلك البكسلات، والطريقة الوحيدة لنقل شكل على لوحة هي بمسح اللوحة أو الجزء الذي يحيط بالشكل، ثم إعادة رسمه بالشكل في موضع جديد.

17.1 الرسوميات المتجهية القابلة للتحجيم SVG

لن يخوض هذا الكتاب في تفاصيل SVG وإنما سنشرح كيفية عملها باختصار، كما سنعود إلى عيوبها في نهاية الفصل، والتي يجب وضعها في حسبانك حين تريد اتخاذ قرار بشأن آلية الرسم المناسبة لتطبيق ما.

فيما يلي مستند HTML مع صورة SVG بسيطة:

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

تغيِّر السمة `xmlns` عنصرًا ما -وعناصره الفرعية- إلى فضاء اسم XML مختلف، حيث يحدِّد ذلك الفضاء المعرَّف بواسطة رابط تشعبي URL الصيغة التي نستخدِمها الآن، وعلى ذلك يكون للوسمين `<circle>` و`<rect>` معنىً هنا في SVG، رغم أنهما لا يمثِّلان شيئًا في لغة HTML، كما يرسمان هنا الأشكال باستخدام التنسيق والموضع اللذين يحدَّدان بواسطة سماتهما.

تنشئ هذه الوسوم عناصر DOM وتستطيع السكربتات أن تتفاعل معها كما تفعل وسوم HTML تمامًا، إذ تغيِّر الشيفرة التالية مثلًا عنصر `<circle>` ليُلوَّن باللون السماوي Cyan:

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

17.2 عنصر اللوحة

يمكن رسم رسوميات اللوحة على عنصر `<canvas>`، كما تستطيع إعطاء مثل هذا العنصر سمات عرض `width` وطول `height` لتحديد حجمها بالبكسلات، وتكون اللوحة الجديدة فارغةً تمامًا، مما يعني أنها شفافة وتظهر مثل مساحة فارغة في المستند، كما يسمح الوسم `<canvas>` بتنسيقات مختلفة من الرسم، ونحتاج إلى إنشاء سياق context أولًا للوصول إلى واجهة الرسم الحقيقية، وهو كائن توفر توابعه واجهة الرسم.

لدينا حاليًا تنسيقَين من أنماط الرسم المدعومَين دعمًا واسعًا هما `"2d"` للرسم ثنائي الأبعاد و`"webgl"` للرسم ثلاثي الأبعاد من خلال واجهة OpenGL، كما أننا لن نناقش واجهة OpenGL هنا، وإنما سنقتصر على الرسم ثنائي الأبعاد، لكن إذا أردت النظر في الرسم ثلاثي الأبعاد فاقرأ في WebGL، إذ توفر واجهةً مباشرةً لعتاد الرسوميات، وتسمح لك بإخراج مشاهد معقدة بكفاءة عالية باستخدَام جافاسكربت.

نستطيع إنشاء سياق بواسطة التابع `getContext` على `<canvas>` لعنصر DOM كما يلي:

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

يرسم المثال مستطيلًا أحمرًا بعرض 100 بكسل وارتفاع 50 بكسل بعد إنشاء كائن السياق، ويكون الركن الأيسر العلوي في الإحداثيات هو (10,10)، كما يضع نظام الإحداثيات في عنصر اللوحة الإحداثيات الصفرية (0,0) في الركن الأيسر العلوي كما في HTML وSVG، بحيث يتجه محور الإحداثي y لأسفل من هناك، وبالتالي يكون (10,10) مزاحًا عشرة بكسلات إلى الأسفل وإلى يمين الركن الأيسر العلوي.

17.3 الأسطر والأسطح

نستطيع ملء الشكل في واجهة اللوحة، مما يعني أننا سنعطي مساحته لونًا أو نقشًا بعينه، أو يمكن تحديده stroked بأن يُرسَم خطًا حول حوافه، وما قيل هنا سيقال في شأن SVG أيضًا، كما يملأ التابع `fillRect` مستطيلًا ويأخذ إحداثيات x وy للركن العلوي الأيسر للمستطيل ثم عرضه ثم ارتفاعه، ويرسم التابع `strokeRect` بالمثل الخطوط الخارجية للمستطيل، لكن لا يأخذ هذان التابعان معاملات أخرى، فلا يحدَّد وسيط ما لون الملء ولا سماكة التحديد ولا غيرها، كما قد يُتوقَّع في مثل هذه الحالة، والذي يحدِّد تلك العناصر هي خصائص سياق الكائن، حيث تتحكم الخاصية `fillStyle` بطريقة ملء الأشكال، ويمكن تعيينها لتكون سلسلةً نصيةً تحدِّد لونًا ما باستخدام ترميز الألوان في CSS؛ أما الخاصية `strokeStyle` فهي شبيهة بأختها السابقة، لكن تحدد اللون المستخدَم في التحديد، كما يُحدَّد عرض الخط بواسطة الخاصية `lineWidth` التي قد تحتوي أي عدد موجب.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

إذا لم تُحدَّد سمة عرض width أو طول height كما في المثال، فسيحصل عنصر اللوحة على عرض افتراضي مقداره 300 بكسل وطول مقداره 150 بكسل.

17.4 المسارات

المسار هو متسلسلة من الأسطر، ويأخذ عنصر اللوحة ثنائي الأبعاد منظورًا استثنائيًا لوصف مثل تلك المسارات من خلال التأثيرات الجانبية بالكامل، كما لا تُعَدّ المسارات قيمًا يمكن تخزينها وتمريرها من مكان إلى آخر، بل إذا أردنا فعل شيء بمسار ما، فيمكن إنشاء متسلسلة من استدعاءات التوابع لوصف شكله.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

ينشئ هذا المثال مسارًا فيه عدد من أجزاء أسطر أفقية ثم يحدِّدها باستخدام التابع stroke، ويبدأ كل جزء أُنشئ بواسطة lineTo عند الموضع الحالي للمسار، كما يكون ذلك الموضع عادةً في نهاية الجزء الأخير، إلا إذا استدعيت moveTo، حيث سيبدأ الجزء التالي في تلك الحالة عند الموضع الممرَّر إلى moveTo.

يُملأ كل شكل لوحده عند ملء المسار باستخدام التابع fill، وقد يحتوي المسار على عدة أشكال، بحيث تبدأ كل حركة moveTo شكلًا جديدًا، ولكن سيحتاج المسار إلى أن يغلَق قبل إمكانية ملئه، بحيث تكون بدايته ونهايته في الموضع نفسه، فإذا لم يكن المسار مغلقًا، فسيضاف السطر من نهايته إلى بدايته، ويُملأ الشكل المغلَّف بالمسار المكتمِل.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
```

```
</script>
```

يرسم هذا المثال مثلثًا مملوءًا. لاحظ أنّ ضلعيَن فقط من أضلاع المثلث هما اللذان رُسما صراحةً؛ أما الثالث الذي يبدأ من الركن السفلي الأيمن إلى القمة فيُضمَّن ولن يكون موجودًا عند تحديد المسار، كما يُستخدَم التابع `closePath` لغلق المسار صراحةً من خلال إضافة جزء سطر حقيقي إلى بداية المسار، وسيُرسم هذا الجزء عند تحديد المسار.

17.5 المنحنيات

قد يحتوي المسار على خطوط منحنية، وتكون هذه الخطوط أعقد في رسمها من الخطوط المستقيمة، حيث يرسم التابع `quadraticCurveTo` منحني إلى نقطة ما، كما يُعطى التابع نقطة تحكم ونقطة وجهة لتحديد انحناء الخط، ويمكن تخيل نقطة التحكم على أنها تسحب الخط لتعطيه انحناءه؛ أما الخط نفسه فلن يمر عليها وإنما سيكون اتجاهه عند نقطتي البدء والنهاية، بحيث إذا رُسم خط مستقيم في ذلك الاتجاه فسيشير إلى نقطة التحكم، ويوضِّح المثال التالي ذلك:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) goal=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

سنرسم المنحنى التربيعي من اليسار إلى اليمين، وتكون نقطة التحكم هي (60,10)، ثم نرسم خطين يمران بنقطة التحكم تلك ويعودان إلى بداية الخط. سيكون الشكل الناتج أشبه بشعار أفلام ستار تريك Star Trek، كما تستطيع رؤية تأثير نقطة التحكم، بحيث تبدأ الخطوط تاركة الأركان السفلى في اتجاه نقطة التحكم ثم تنحني مرةً أخرى إلى هدفها.

يرسم التابع `bezierCurveTo` انحناءً قريبًا من ذلك، لكن يكون له نقطتي تحكم أي واحدة عند كل نهاية خط بدلًا من نقطة تحكم واحدة، ويوضِّح المثال التالي سلوك هذا المنحنى:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10,10) control2=(90,10) goal=(50,90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

تحدد نقطتا التحكم الاتجاه عند كلا النهايتين للمنحنى، وكلما ابتعدنا عن النقطة الموافقة لهما زاد انتفاخ المنحنى في ذلك الاتجاه، كما ستكون تلك المنحنيات أصعب من حيث العمل عليها، فليس من السهل معرفة كيفية إيجاد نقاط التحكم التي توفر الشكل الذي تبحث عنه، حيث نستطيع حسابها أحيانًا، لكن سيكون علينا إيجاد قيمة مناسبة من خلال التجربة والخطأ أحيانًا أخرى.

يُستخدَم التابع `arc` على أساس طريقة لرسم خط ينحني على حواف دائرة، كما يأخذ زوجًا من الإحداثيات من أجل مركز القوس، ونصف قطر، ثم زاوية بداية وزاوية نهاية.

نستطيع من خلال هذين المعاملَين الأخيرين رسم جزء من الدائرة فقط دون رسمها كلها، كما تقاس الزوايا بالراديان radian وليس بالدرجات، ويعني هذا أن الدائرة الكاملة لها زاوية مقدارها `2π` أو `2 * Math.PI`، وهي تساوي 6.28 تقريبًا، كما تبدأ الزاوية العد عند النقطة التي على يمين مركز الدائرة وتدور باتجاه عقارب الساعة من هناك، وهنا تستطيع استخدام 0 للبداية ونهاية تكون أكبر من 2π -لتكن 7 مثلًا- من أجل رسم الدائرة كلها.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to ½π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

ستحتوي الصورة الناتجة على خط من يمين الدائرة الكاملة -أي أول استدعاء إلى `arc`- إلى يمين ربع الدائرة -أي الاستدعاء الثاني-، حيث يتصل الخط المرسوم بواسطة `arc` بقطعة المسار السابقة كما في توابع رسم المسارات الأخرى، كما تستطيع هنا استدعاء `moveTo` أو البدء في مسار جديد لتجنب هذا.

17.6 رسم المخطط الدائري

لنقل أننا نريد رسم مخطط دائري لنتائج استبيان رضا العملاء عن شركة ما ولتكن EconomiCorp مثلًا، بحيث تحتوي رابطة `results` على مصفوفة من الكائنات التي تمثل نتائج الاستبيان.

```
const results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
  {name: "Unsatisfied", count: 510, color: "pink"},
  {name: "No comment", count: 175, color: "silver"}
];
```

سنرسم عددًا من الشرائح الدائرية يتكون كل منها من قوس وزوج من الخطوط ينتهيان إلى مركز ذلك القوس، كما نستطيع حساب الزاوية التي يأخذها كل قوس من خلال قسمة الدائرة الكلية 2π على العدد الكلي للاستجابات، ومن ثم ضرب ذلك العدد -زاوية الاستجابة- في عدد الأشخاص الذين اختاروا عنصرًا ما.

```
<canvas width="200" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  // ابدأ من القمة
  let currentAngle = -0.5 * Math.PI;
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    // center=100,100, radius=100
    // من الزاوية الحالية، باتجاه عقارب الساعة بحذاء زاوية الشريحة
    cx.arc(100, 100, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
```

```
    cx.fill();
  }
</script>
```

لا يخبرنا المخطط ماذا تعني تلك الشرائح، لذا سنحتاج إلى طريقة نرسم بها نصوصًا على اللوحة.

17.7 النصوص

يوفر سياق رسم اللوحة ثنائي الأبعاد التابعَين `fillText` و`strokeText`، حيث يُستخدَم الأخير في تحديد الأحرف، لكن الذي نحتاج إليه هو `fillText` عادةً، إذ سيملأ حد النص المعطى بتنسيق `fillStyle` الحالي.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>
```

تستطيع تحديد حجم النص وتنسيقه ونوع خطه أيضًا باستخدام الخاصية `font`، ولا يعطينا هذا المثال إلا حجم الخط واسم عائلته، كما من الممكن إضافة ميل الخط `italic` أو سماكته `bold` إلى بداية السلسلة النصية لاختيار تنسيق ما، في حين يوفر آخر وسيطين لكل من `fillText` و`strokeText` الموضع الذي سيُرسم فيه الخط، كما يشيران افتراضيًا إلى موضع بداية قاعدة النص الأبجدية التي تكوِّن السطر الذي تقف الحروف عليه، لكن لا تحسب الأجزاء المتدلية من الأحرف مثل حرف j أو p، ونستطيع تغيير الموضع الأفقي ذاك بضبط الخاصية `textAlign` لتكون `"end"` أو `"center"`، وتغيير الموضع الرأسي كذلك من خلال ضبط `textBaseline` لتكون `"top"` أو `"middle"` أو `"bottom"`.

17.8 الصور

يُفرَّق عادةً في رسوميات الحواسيب بين الرسوميات المتجهية vector graphics والرسوميات النقطية bitmap graphics، فالأولى هي التي شرحناها في بداية هذا الفصل والتي تصف الصورة وصفًا منطقيًا لشكلها؛ أما الرسوميات النقطية فلا تصف الأشكال الحقيقية، بل تعمل مع بيانات البكسلات الخاصة بالصورة والتي هي مربعات من النقاط الملونة على الشاشة.

يسمح لنا التابع `drawImage` برسم بيانات البكسلات على اللوحة، ويمكن استخراج تلك البيانات من عنصر `<img>` أو من لوحة أخرى، كما ينشئ المثال التالي عنصر `<img>` منفصل ويحمِّل ملف الصورة إليه، لكنه

لا يستطيع البدء بالرسم مباشرةً من تلك الصورة بما أنّ المتصفح قد لا يكون حمَّلها بعد، ولحل هذا فإننا نسجل معالج الحدث `"load"` لتنفيذ الرسم بعد تحميل الصورة.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>
```

سيرسم التابع `drawImage` الصورة بحجمها الحقيقي افتراضيًا، لكن يمكن إعطاؤه وسيطَين إضافيين لتحديد عرض وطول مختلفَين، فإذا أعطي `drawImage` تسعة وسائط، فيمكن استخدامه لرسم جزء من الصورة، ويوضِّح الوسيط الثاني حتى الخامس -أي x وy والعرض والطول- المستطيل الذي في الصورة المصدرية والتي يجب نسخها؛ أما الوسيط السادس حتى التاسع فتعطينا المستطيل على اللوحة الذي سيُنسخ، كما يمكن استخدام هذا لتحزيم عدة شرائح أو عناصر من صورة (تسمى sprites أي عفاريت) في ملف صورة واحد، ثم رسم الجزء الذي نحتاج إليه فقط، فلدينا مثلًا هذه الصورة التي تحتوي على شخصية لعبة في عدة وضعيات:

إذا غيرنا الوضع الذي نرسمه فسنستطيع عرض تحريك يبدو مثل شخصية تمشي، في حين نستخدِم التابع `clearRect` لتحريك صورة على اللوحة، وهو يمثِّل `fillRect`، لكنه يجعل المستطيل شفافًا بدلًا من تلوينه حاذفًا البكسلات المرسومة سابقًا، ونحن نعرف أنّ كل عفريت وكل صورة فرعية يكون عرضها 24 بكسل وارتفاعها 30 بكسل، وعلى ذلك تحمِّل الشيفرة التالية الصورة ثم تضبط فترةً زمنيةً متكررةً لرسم الإطار التالي:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
```

```
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // المستطيل المصدر
                   cycle * spriteW, 0, spriteW, spriteH,
                   // مستطيل الوجهة
                   0,               0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

تتعقب رابطة `cycle` موضعنا في هذا التحريك وتتزايد مرةً لكل إطار، ثم تقفز عائدةً إلى المجال 0 إلى 7 باستخدام عامِل الباقي، بعدها تُستخدَم هذه الرابطة بعد ذلك لحساب الإحداثي x الذي يحتوي عليه العفريت الذي في الوضع الحالي في الصورة.

17.9 التحول

ماذا لو أردنا جعل الشخصية تمشي إلى اليسار بدلًا من اليمين؟ لا شك أننا نستطيع رسم مجموعة أخرى من عناصر العفاريت، لكننا نستطيع توجيه اللوحة أيضًا لترسم الصورة بعكس الطريقة التي ترسمها بها، كما سيزيد استدعاء التابع `scale` حجم أيّ شيء يُرسم بعده، وهو يأخذ معاملَين أحدهما لضبط المقياس الأفقي والآخر للعمودي.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

سيتسبب تغيير حجم الصورة في تمديد كل شيء فيها أو ضغطه بما فيها عرض الخط، وإذا غيّرنا المقياس ليكون بقيمة سالبة، فستنقلب الصورة معكوسة، حيث يحدث الانعكاس حول النقطة (0,0) التي تعني أننا سنقلب أيضًا اتجاه نظام الإحداثيات، فحين نطبّق مقياسًا أفقيًا مقداره `-1`، فسيكون الشكل المرسوم عند الموضع 100 على إحداثي x في الموضع الذي كان `-100` من قبل، لذا لا نستطيع إضافة `cx.scale(-1, 1)` من أجل عكس الصورة وحسب قبل استدعاء `drawImage`، لأنه سيجعل الصورة تتحرك خارج اللوحة بحيث تكون غير مرئية، ونعدّل الإحداثيات المعطاة إلى `drawImage` من أجل ضبط هذا برسم الصورة في الموضع `-50` على الإحداثي x بدلًا من 0.

هناك حل آخر لا يحتاج إلى الشيفرة التي تنفذ الرسم كي يدرك تغير المقياس، وهو تعديل المحور الذي يحدث تغيير الحجم حوله، كما يمكن استخدام عدة توابع أخرى غير `scale` للتأثير في نظام إحداثيات اللوحة، حيث تستطيع تدوير الأشكال المرسومة تاليًا باستخدام التابع `rotate` ونقلها باستخدام `translate`، لكن المثير في الأمر والمحير أيضًا هو أنّ تلك التحويلات تُكدَّس، بمعنى أنّ كل واحد يُحدِث نسبةً إلى ما قبله من تحولات، وبناءً عليه فإذا استخدمنا `translate` لتحريك 10 بكسلات مرتين أفقيًا، فسيُرسم كل شيء مزاحًا إلى اليمين بمقدار 20 بكسل؛ أما إذا أزحنا مركز نظام الإحداثيات أولًا إلى (50,50) ثم دوّرنا بزاوية 20 درجة -أي 0.1π راديان-، فسيَحدث التدوير حول النقطة (50,50).

لكن إذا نفذّنا التدوير بمقدار عشرين درجة أولًا ثم أزحنا بمقدار (50,50)، فسيحدث الإزاحة عند نظام الإحداثيات المدوَّر، وعليه سيعطينا اتجاهًا مختلفًا، ونستنتج من هذا أنّ ترتيب تطبيق التحويلات مهم. وتعكس الشيفرة التالية الصورة حول الخط العمودي عند الموضع x:

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

ننقل المحور y إلى حيث نريد لمرآتنا أن تكون ونطبّق المرآة، ثم نعيد المحور مرةً أخرى إلى موضعه المناسب في العالم المعكوس، وتوضِّح الصورة التالية ذلك:

توضِّح الصورة نظام الإحداثيات قبل وبعد الانعكاس على طول الخط المركزي وتُرقَّم المثلثات لتوضيح كل خطوة، فإذا رسمنا مثلثًا عند الموضع x الموجب، فسيكون حيث يكون المثلث 1، إذ نستدعي `flipHorizontally` لينفِّذ الإزاحة إلى اليمين أولًا لنصل إلى المثلث 2، ثم يغيِّر الحجم ويعكس المثلث إلى الموضع 3، غير أنه لا يُفترض أن يكون هناك إذا عُكِس في الخط المعطى، فيأتي استدعاء `translate` الثاني ليصلح ذلك، بحيث يلغي الإزاحة الأولى ويُظهِر المثلث 4 في الموضع الذي يُفترض أن يكون فيه تمامًا، ونستطيع الآن رسم الشخصية المعكوسة في الموضع (100,0) من خلال عكس العالم حول المركز العمودي للشخصية.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

17.10 تخزين التحويلات ومحوها

تظل التحويلات قائمةً حتى بعد رسمنا لتلك الشخصية المعكوسة، إذ سيكون كل شيء نرسمه بعد ذلك معكوسًا أيضًا وقد لا نريد هذا، حيث من الممكن هنا حفظ التحويل الحالي ثم إجراء بعض الرسم والتحويل، وبعدها استعادة التحويل القديم مرةً أخرى، فغالبًا يُعَدّ ما سبق الإجراء الأفضل لإجراء وظيفة دالة ما تحتاج إلى

تحويل نظام الإحداثيات لفترة مؤقتة، حيث سنحفظ أيّ تحويل استخدمته الشيفرة التي استدعت الدالة، ثم تضيف الدالة ما تشاء من التحويلات فوق التحويل الحالي، وبعد ذلك نرجع إلى التحول الذي بدأنا به مرةً أخرى.

يدير عملية التحول تلك التابعَين save وrestore على سياق اللوحة ثنائية الأبعاد لأنهما يحتفظان بمكدس من حالات التحول، وحين نستدعي save، فستُدفَع الحالة الراهنة إلى المكدس، ثم تؤخَذ الحالة التي على قمة المكدس مرةً أخرى عند استدعاء restore وتُستخدم على أنها التحول الحالي للسياق، كما نستطيع كذلك استدعاء resetTransform من أجل إعادة ضبط التحول بالكامل.

توضِّح الدالة branch في المثال التالي ما يمكن فعله بدالة تغير التحول، ثم تستدعي دالةً -هي نفسها في هذه الحالة- تتابع الرسم بالتحول المعطى، حيث ترسم تلك الدالة شكلًا يشبه الشجرة من خلال رسم خط ثم نقل مركز نظام الإحداثيات إلى نهاية ذلك الخط، ثم استدعاء نفسها مرتين، مرةً مدارةً إلى اليسار، ثم مرةً أخرى مدارة إلى اليمين، إذ يقلل كل استدعاء من طول الفرع المرسوم ثم يتوقف التعاود حين يقل الطول عن 8.

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

إذا لم توجد استدعاءات إلى save وrestore فسيكون موضع ودران الاستدعاء الذاتي للمرة الثانية للدالة branch هما اللذان أُنشئا بالاستدعاء الأول، إذ لن تتصل بالفرع الحالي وإنما بالفرع الداخلي الأقصى إلى اليمين والمرسوم بالاستدعاء الأول، وعلى ذلك يكون الشكل الناتج ليس شجرةً أبدًا.

17.11 عودة إلى اللعبة

عرفنا الآن ما يكفي عن الرسم على اللوحة وسنعمل الآن على نظام عرض مبني على لوحة من أجل اللعبة التي من الفصل السابق، إذ لم تعُد الشاشة الجديدة تعرض صناديق ملونة فحسب، وإنما سنستخدِم `drawImage` من أجل رسم صور تمثِّل عناصر اللعبة، كما سنعرِّف كائن عرض جديد يدعى `CanvasDisplay` ويدعم الواجهة نفسها مثل `DOMDisplay` من الفصل السادس عشر خاصةً التابعَين `syncState` و`clear`، حيث سيحتفظ هذا الكائن بمعلومات أكثر قليلًا من `DOMDisplay`، فبدلًا من استخدام موضع التمرير لعنصر DOM الخاص به، فهو يتتبع نافذة رؤيته التي تخبرنا بالجزء الذي ننظر إليه في المستوى، ثم يحتفظ بالخاصية `flipPlayer` التي تمكّن اللاعب من مواجهة الاتجاه الذي كان يتحرك فيه حتى لو كان واقفًا لا يتحرك.

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

يحسب التابع syncState أولًا نافذة رؤية جديدة ثم يرسم مشهد اللعبة عند الموضع المناسب.

```
CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};
```

سيكون على هذا التنسيق من العرض إعادة رسم الخلفية عند كل تحديث على عكس DOMDisplay، ولأن الأشكال التي على اللوحة ما هي إلا بكسلات، فليس لدينا طريقةً لتحريكها أو حتى حذفها بعد رسمها، والطريقة الوحيدة لدينا لتحديث شاشة اللوحة هي مسحها ثم إعادة رسم المشهد.، وقد يحدث أن نكون قد مرّرنا نافذة الرؤية من الشاشة، وهذا يتطلب أن تكون الخلفية في موضع مختلف.

يتحقق التابع updateViewport إذا كان اللاعب قريبًا للغاية من حافة الشاشة أم لا، وإذا كان كذلك فسينقل نافذة الرؤية، وهو في هذا يشبه التابع scrollPlayerIntoView الخاص بـ DOMDisplay.

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
                         state.level.width - view.width);
  }
  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
                        state.level.height - view.height);
  }
};
```

تتأكد الاستدعاءات إلى Math.max وMath.min من أنّ نافذة الرؤية لا تعرض مساحةً خارج المستوى، حيث تضمن Math.max(x, 0) أنّ العدد الناتج ليس أقل من صفر، كما تضمن Math.min بقاء القيمة

تحت الحد المعطى، وسنستخدم عند مسح الشاشة لونًا مختلفًا وفقًا لحالة اللعب إذا فازت أو خسرت، بحيث يكون لونًا فاتحًا في حالة الفوز، وداكنًا في الخسارة.

```
CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
    this.cx.fillStyle = "rgb(68, 191, 255)";
  } else if (status == "lost") {
    this.cx.fillStyle = "rgb(44, 136, 214)";
  } else {
    this.cx.fillStyle = "rgb(52, 166, 251)";
  }
  this.cx.fillRect(0, 0,
                   this.canvas.width, this.canvas.height);
};
```

نمر على المربعات المرئية في نافذة الرؤية الحالية من أجل رسم الخلفية باستخدام الطريقة نفسها التي اتبعناها في التابع `touches` في الفصل السابق.

```
let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? scale : 0;
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
```

```
                                screenX, screenY, scale, scale);
      }
    }
  };
```

ستُرسم المربعات غير الفارغة باستخدام drawImage، وتحتوي الصورة otherSprites على الصور المستخدَمة للعناصر سوى اللاعب، فهي تحتوي من اليسار إلى اليمين على مربع الحائط ومربع الحمم البركانية وعفريت للعملة.

تكون مربعات الخلفية بقياس 20×20 بكسل بما أننا سنستخدم نفس المقياس الذي استخدمناه في DOMDisplay، وعلى ذلك ستكون إزاحة مربعات الحمم البركانية هي 20 -وهي قيمة رابطة scale-؛ أما إزاحة الجدران فستكون صفرًا، كما لن ننتظر تحميل عفريت الصورة لأنّ استدعاء drawImage بصورة لم تحمَّل بعد فلن يُحدث شيئًا، وبالتالي فقد نفشل في رسم اللعبة في أول بضعة إطارات أثناء تحميل تلك الصورة لكنها ليست تلك بالمشكلة الكبيرة، وسيظهر المشهد الصحيح بمجرد انتهاء التحميل بما أننا نظل نحدث الشاشة.

ستُستخدم الشخصية الماشية التي رأيناها سابقًا لتمثل اللاعب، ويجب على الشيفرة التي ترسمها اختيار العفريت الصحيح والاتجاه الصحيح كذلك لحركة اللاعب الآنية، إذ تحتوي أول ثمانية عفاريت على تأثير المشي، كما نكرر تلك العناصر عند مشي اللاعب على الأرضية وفقًا للزمن الحالي، وبما أننا نبدِّل الإطارات كل 60 ميلي ثانية، فسنقسم الوقت على 60 أولًا؛ أما حين يقف اللاعب ساكنًا فسنرسم العفريت التاسع، ونستخدِم العنصر العاشر من أقصى اليمين من أجل رسم تأثير القفز الذي نعرفه من خلال كون السرعة العمودية لا تساوي صفرًا.

يجب على التابع تعديل إحداثيات x والعرض بمقدار معطى (playerXOverlap) لمعادلة عرض العفاريت بما أنها أعرض من كائن اللاعب -24 بكسل بدلًا من 16- لتسمح ببعض المساحة للأذرع والأقدام.

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0) {
```

```
      this.flipPlayer = player.speed.x < 0;
    }

    let tile = 8;
    if (player.speed.y != 0) {
      tile = 9;
    } else if (player.speed.x != 0) {
      tile = Math.floor(Date.now() / 60) % 8;
    }

    this.cx.save();
    if (this.flipPlayer) {
      flipHorizontally(this.cx, x + width / 2);
    }
    let tileX = tile * width;
    this.cx.drawImage(playerSprites, tileX, 0, width, height,
                                     x,     y, width, height);
    this.cx.restore();
  };
```

يُستدعى التابع `drawPlayer` بواسطة `drawActors` التي تكون مسؤولةً عن رسم جميع الكائنات الفاعلة في اللعبة.

```
  CanvasDisplay.prototype.drawActors = function(actors) {
    for (let actor of actors) {
      let width = actor.size.x * scale;
      let height = actor.size.y * scale;
      let x = (actor.pos.x - this.viewport.left) * scale;
      let y = (actor.pos.y - this.viewport.top) * scale;
      if (actor.type == "player") {
        this.drawPlayer(actor, x, y, width, height);
      } else {
        let tileX = (actor.type == "coin" ? 2 : 1) * scale;
        this.cx.drawImage(otherSprites,
                          tileX, 0, width, height,
                          x,     y, width, height);
      }
```

```
    }
  };
```

عند رسم أيّ شيء غير اللاعب فإننا ننظر أولًا في نوعه لنعرف إزاحة العفريت الصحيح، فمربع الحمم إزاحته 20، اما عفريت العملة فإزاحته 40، أي أنه ضعف مقدار `scale`، كما يجب طرح موضع نافذة الرؤية على لوحتنا لتتوافق مع أعلى يسار نافذة الرؤية وليس أعلى يسار المستوى، كما يمكن استخدام `translate` كذلك، فكلاهما يصلح.

يركِّب المستند التالي الشاشة الجديدة بـ `runGame`:

```
<body>
  <script>
    runGame(GAME_LEVELS, CanvasDisplay);
  </script>
</body>
```

17.12 اختيار واجهة الرسوميات

لدينا عدة خيارات يمكن استخدامها لتوليد الرسوميات في المتصفح من HTML إلى SVG إلى اللوحة، وليس هناك واحد يفضُلها جميعًا في كل حالة، فكل واحد له نقاط قوته وضعفه، فلغة HTML مثلًا تمتاز بالبساطة، كما تتكامل جيدًا مع النصوص، بينما تسمح لك كل من SVG واللوحة برسم النصوص، لكنها لن تمكّنك من موضعة تلك النصوص أو تغليفها إذا أخذت أكثر من سطر واحد؛ أما في الصور المبنية على HTML فسيكون من السهل إدراج كتل نصية فيها.

يمكن استخدام SVG من ناحية أخرى لإنتاج رسوميات واضحة مهما كان مستوى التكبير، فهي مصممة للرسم على عكس HTML، وعليه فهي ملائمة أكثر لهذا الغرض، كذلك تستطيع كل من SVG وHTML بناء هيكل بيانات مثل شجرة DOM تمثل صورتنا، وهذا يمكِّننا من تعديل العناصر بعد رسمها، لذا سيكون من الصعب استخدام اللوحة من أجل تغيير جزء صغير في صورة كبيرة كل حين للاستجابة لأفعال المستخدِم أو بسبب تحريك ما، كما يسمح DOM لنا بتسجيل معالجات أحداث الفأرة على كل عنصر في الصورة حتى الأشكال المرسومة باستخدام SVG؛ أما اللوحة فلا يمكن فعل ذلك فيها.

غير أنه يمكن استخدام نهج المنظور البكسلي pixel-oriented للوحة عند رسم أعداد كبيرة جدًا من عناصر صغيرة، إذ تكون تكلفة الشكل الواحد فيها تافهةً بما أنها لا تبني هياكل بيانات وإنما تكرِّر الرسم على مساحة البكسل نفسها، كما يمكن تنفيذ تأثيرات مثل إخراج مشهد بسرعة بكسل واحد في كل مرة -باستخدام متعقب أشعة ray tracer مثلًا-، أو معالجة لاحقة لصورة باستخدام جافاسكربت مثل تأثير الضباب أو التشويش، وذلك لا يمكن معالجته بواقعية إلا من المنظور البكسلي.

قد نرغب أحيانًا في جمع بعض تلك التقنيات معًا، فقد نرسم مخططًا باستخدام SVG أو اللوحة، لكن نُظهر المعلومات النصية عن طريق وضع عنصر HTML فوق الصورة؛ أما بالنسبة للتطبيقات التي لا تتطلب موارد كثيرة، فليس من المهم أيّ واجهة نختارها، إذ يمكن استخدام العرض الذي بنيناه في هذا الفصل من أجل لعبتنا، كان يمكن تنفيذه باستخدام أي من التقنيات الرسومية الثلاثة بما أنه لا يحتاج إلى رسم نصوص أو معالجة تفاعلات للفأرة أو التعامل مع عدد ضخم من العناصر.

17.13 خاتمة

لقد ناقشنا في هذا الفصل تقنيات رسم التصاميم المرئية والرسوميات في المتصفح مع تناول عنصر `<canvas>` بالتفصيل، كما عرفنا أنّ عقدة اللوحة تمثِّل مساحةً في مستند قد يرسم برنامجنا عليها، وينفَّذ هذا الرسم من خلال رسم كائن سياقي ينشئه التابع `getContext`، كما تسمح لنا واجهة الرسم ثنائية الأبعاد بملء وتخطيط أشكال كثيرة، وتحدد خاصية السياق `fillStyle` كيفية ملء الأشكال، في حين تتحكم الخاصيتان `strokeStyle` و`lineWidth` في طريقة رسم الخطوط.

تُرسم المستطيلات وأجزاء النصوص باستدعاء تابع واحد، حيث يرسم التابعان `fillRect` و`strokeRect` مستطيلات، بينما يرسم كل من `fillText` و`strokeText` نصوصًا؛ أما إذا أردنا إنشاء أشكال فيجب علينا بناء مسار أولًا، كما ينشئ استدعاء `beginPath` مسارًا جديدًا، كما يمكن إضافة خطوط ومنحنيات إلى المسار الحالي باستخدام عدة توابع أخرى، حيث يضيف التابع `lineTo` مثلًا خطًا مستقيمًا، وإذا انتهى المسار، فيمكن استخدام التابع `fill` لملئه أو التابع `stroke` لتحديده.

تُنقل البكسلات من صورة أو لوحة أخرى إلى لوحتنا باستخدام التابع `drawImage`، حيث يرسم هذا التابع الصورة المصدرية كلها افتراضيًا، لكن يمكن إعطاؤه معاملات إضافية من أجل نَسخ جزء محدد من الصورة، وقد استخدَمنا ذلك في لعبتنا بنسخ أوضاع شخصية اللاعب المختلفة من صورة تحتوي على عدة أوضاع.

تسمح لنا التحولات برسم شكل في اتجاهات مختلفة، فسياق الرسم ثنائي الأبعاد به تحول راهن يمكن تغييره باستخدام التوابع `translate` و`scale` و`rotate`، إذ ستؤثِّر على جميع عمليات الرسم اللاحقة، كما يمكن حفظ حالة التحول باستخدام التابع `save` واستعادتها باستخدام التابع `restore`، وأخيرًا يُستخدَم التابع `clearRect` عند عرض تحريك على اللوحة من أجل مسح جزء من اللوحة قبل إعادة رسمه.

17.14 تدريبات

17.14.1 الأشكال

اكتب برنامجًا يرسم الأشكال التالية على لوحة:

1. شبه منحرف وهو مستطيل أحد جوانبه المتوازية أطول من الآخر.

2. ماسة حمراء وهي مستطيل مُدار بزاوية 45 درجة مئوية، أو π¼ راديان.
3. خط متعرِّج Zigzag.
4. شكل حلزوني من 100 جزء من خطوط مستقيمة.
5. نجمة صفراء.

ربما تود الرجوع إلى شرح كل من `Math.cos` و`Math.sin` في الفصل الرابع عشر عند رسم آخر شكلين لتعرف كيف تحصل على إحداثيات على دائرة باستخدام هاتين الدالتين، كما ننصحك بإنشاء دالة لكل شكل وتمرير الموضع إليها و-إذا أردت- مرر إليها الخصائص الأخرى مثل الحجم وعدد النقاط على أساس معاملات، كما يوجد حل بديل بأن تكتب أرقامًا ثابتةً في شيفرتك مما سيجعل الشيفرة أصعب في القراءة والتعديل.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<canvas width="600" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");

  // شيفرتك هنا.
</script>
```

إرشادات الحل

يمكن رسم شبه المنحرف ببساطة باستخدام مسار، لذا اختر إحداثيات مركزية مناسبة وأضف كل ركن من الأركان الأربعة حول المركز؛ أما الماسة فيمكن رسمها بطريقة بسيطة باستخدام مسار أو باستخدام تحول `rotate`، وإذا أردت تجربة التحول فسيكون عليك تطبيق طريقة تشبه ما فعلنا في دالة `flipHorizontally`، كما يجب أن تنتقل `translate` إلى مركز المستطيل بدلًا من الإحداثي الصفري بما أنك ستدور حول مركزه، ثم تدور ثم تُعاد إلى مكانها مرةً أخرى، وتأكد من إعادة ضبط التحول بعد رسم أيّ شكل ينشئ تحولًا.

أما بالنسبة للخط المتعرِّج فليس من المنطقي كتابة استدعاء جديد إلى `lineTo` في كل جزء من أجزاء الخط، وإنما يجب استخدام حلقة تكرارية، بحيث تجعل كل تكرار فيها يرسم جزأين -اليمين ثم اليسار مرةً أخرى-

أو جزءًا واحدًا من أجل تحديد اتجاه الذهاب إلى اليمين أم اليسار والذي يكون بالاعتماد على عامل حالة من فهرس الحلقة `i` (مثل إذا كان `i % 2 == 0` اذهب لليسار وإلا، لليمين)، كما ستحتاج إلى حلقة تكرارية من أجل الشكل الحلزوني، فإذا رسمت سلسلةً من النقاط تتحرك فيها كل نقطة إلى الخارج على دائرة حول مركز الشكل فستحصل على دائرة؛ أما إذا استخدمت الحلقة التكرارية وغيرت نصف قطر الدائرة التي تضع النقطة الحالية عليها ونفّذت عدة حركات فستحصل على شكل حلزوني.

تُبنى النجمة المصورة من خطوط `quadraticCurveTo`، لكن يمكن رسمها باستخدام الخطوط المستقيمة أيضًا من خلال تقسيم دائرة إلى ثمانية أجزاء إذا أردت رسم نجمة بثمانية نقاط، أو إلى أي عدد من الأجزاء تريد، ثم ارسم خطوطًا بين تلك النقاط لتجعلها تنحني نحو مركز النجمة؛ أما إذا استخدمت `quadraticCurveTo`، فستستطيع جعل المركز على أساس نقطة تحكم.

17.14.2 المخطط الدائري

رأينا في هذا الفصل مثالًا لبرنامج يرسم مخططًا دائريًا، عدّل هذا البرنامج ليظهر اسم كل تصنيف بجانب الشريحة التي تمثله، وحاول إيجاد طريقة مناسبة مرئيًا لموضعة ذلك النص تلقائيًا بحيث تصلح لأنواع البيانات الأخرى أيضًا، كما تستطيع الافتراض أنّ التصانيف كبيرة بحيث تترك مساحةً كافيةً لعناوينها، وقد تحتاج إلى دالتي `Math.sin` و`Math.cos` مرةً أخرى من الفصل الرابع عشر.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  let currentAngle = -0.5 * Math.PI;
  let centerX = 300, centerY = 150;

  // أضف شيفرة لرسم عناوين الشرائح في هذه الحلقة
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    cx.arc(centerX, centerY, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
```

```
    cx.lineTo(centerX, centerY);
    cx.fillStyle = result.color;
    cx.fill();
  }
</script>
```

إرشادات الحل

ستحتاج إلى استدعاء `fillText` وضبط خصائص السياق `textAlign` و`textBaseline` بطريقة تجعل النص يكون حيث تريد، وستكون الطريقة المناسبة لموضعة العناوين هي وضع النصوص على الخطوط الذاهبة من مركز المخطط إلى منتصف الشريحة، كما لا تريد وضع النصوص مباشرةً على جانب المخطط على بعد مقدار ما من البكسلات، وتكون زاوية هذا الخط هي `currentAngle + 0.5 * sliceAngle`، كما تبحث الشيفرة التالية عن موضع عليه بحيث يكون على بعد 120 بكسل من المركز:

```
let middleAngle = currentAngle + 0.5 * sliceAngle;
let textX = Math.cos(middleAngle) * 120 + centerX;
let textY = Math.sin(middleAngle) * 120 + centerY;
```

أما بالنسبة لـ `textBaseline`، فإنّ القيمة `"middle"` مناسبة عند استخدام ذلك المنظور، إذ يعتمد ما تستخدِمه لـ `textAlign` على الجانب الذي تكون فيه من الدائرة، فإذا كنت على اليسار، فيجب أن تكون `"right"` والعكس بالعكس، وذلك كي يكون موضع النص بعيدًا عن الدائرة.

إذا لم تعرف كيف تجد الجانب الذي عليه زاوية ما من الدائرة، فانظر في شرح `Math.cos` في الفصل الرابع عشر، إذ يخبرك جيب التمام cosine لتلك الدالة بالإحداثي x الموافق لها، والذي يخبرنا بدوره على أي جانب من الدائرة نحن.

17.14.3 الكرة المرتدة

استخدام تقنية `requestAnimationFrame` التي رأيناها في الفصلين الرابع عشر والسادس عشر من أجل رسم صندوق فيه كرة مرتدة تتحرك بسرعة ثابتة وترتد عن جوانب الصندوق عندما تصطدم بها.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<canvas width="400" height="400"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
```

```
    let lastTime = null;
    function frame(time) {
      if (lastTime != null) {
        updateAnimation(Math.min(100, time - lastTime) / 1000);
      }
      lastTime = time;
      requestAnimationFrame(frame);
    }
    requestAnimationFrame(frame);

    function updateAnimation(step) {
      // شيفرتك هنا.
    }
  </script>
```

إرشادات الحل

يسهل رسم الصندوق باستخدام `strokeRect`، لذا عرِّف رابطةً تحمل حجمه أو عرِّف رابطتَين إذا كان عرض الصندوق يختلف عن طوله؛ أما لإنشاء كرة مستديرة، فابدأ مسارًا واستدعي `arc(x, y, radius, 0, 7)` الذي ينشئ قوسًا من الصفر إلى أكثر من دائرة كاملة ثم املأ المسار.

استخدِم الصنف `Vec` من الفصل السادس عشر ليصف نموذج سرعة الكرة وموضعها، وأعطه سرعةً ابتدائيةً يفضَّل أن تكون عموديةً فقط أو أفقيةً فقط، ثم اضرب تلك السرعة بمقدار الوقت المنقضي لكل إطار، وحين تقترب الكرة جدًا من حائط عمودي، اعكس المكوِّن x في سرعتها، وبالمثل اعكس المكوِّن y إذا اصطدمت بحائط أفقي، وبعد إيجاد موضع الكرة وسرعتها الجديدين، استخدم `clearRect` لحذف المشهد وإعادة رسمه باستخدام الموضع الجديد.

17.14.4 الانعكاس المحسوب مسبقا

إنّ أحد عيوب التحولات أنها تبطئ عملية الرسومات النقطية bitmaps، إذ يجب تحويل موضع وحجم كل بكسل، ويتسبب ذلك في زيادة كبيرة في وقت الرسم في المتصفحات، غير أنها لا تمثِّل تلك مشكلةً في لعبتنا التي نرسم فيها عفريتًا واحدًا؛ أما رسم مئات الشخصيات أو آلاف الجزيئات التي تدور في الهواء نتيجة انفجار مثلًا، فستكون تلك معضلةً حقيقيةً، رغم أنّ المتصفحات قد تعالج ذلك البطء في التحولات مستقبلًا.

فكِّر في طريقة تسمح برسم شخصية معكوسة دون تحميل ملفات صور إضافية، ودون الحاجة إلى إنشاء استدعاءات `drawImage` متحولة لكل إطار.

إرشادات الحل

تستطيع استخدام عنصر اللوحة على أساس صورة مصدرية عند استخدام `drawImage`، حيث يمكن إنشاء عنصر `<canvas>` آخر دون إضافته إلى المستند وسحب عناصر العفاريت المعكوسة إليه مرةً واحدةً، وعند رسم إطار حقيقي ننسخ تلك المعكوسة إلى اللوحة الأساسية، لكن يجب توخي الحذر لأن الصور لا تحمَّل فورًا، فنحن ننفِّذ الرسم المعكوس مرةً واحدةً فقط، وإذا نفَّذناه قبل تحميل الصورة، فلن ترسم أي شيء.

يمكن استخدام معالج `"load"` على الصورة لرسم الصور المعكوسة في لوحة إضافية تُستخدم كمصدر رسم مباشرةً، وستكون فارغًا إلى أن نرسم الشخصية عليها.

دورة إدارة تطوير المنتجات

تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق
وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن

18. طلبيات HTTP والاستمارات

> يجب أن تكون الاتصالات ذات طبيعة عديمة الحالة، إذ يجب أن يحتوي كل طلب من العميل إلى الخادم على جميع المعلومات الضرورية لفهم الطلب، دون الحاجة إلى تخزين أي بيانات على الخادم.
>
> — روي فيلدينج Roy Fielding، كتاب الأنماط المعمارية وتصميم معماريات البرمجيات المبنية على الشبكات Architectural Styles and the Design of Network-based Software Architectures.

يُعَدّ بروتوكول نقل النصوص الفائقة Hypertext Transfer Protocol الذي ذكرناه في الفصل الثالث عشر آليةً تُطلب البيانات وتوفَّر من خلالها على الشبكة العالمية، كما سننظر فيه بالتفصيل ونشرح الطريقة التي تستخدِمه بها جافاسكربت المتصفحات.

18.1 البروتوكول

إذا كتبت eloquentJavaScript.net/18_http.html في شريط العنوان لمتصفحك، فسيبحث المتصفح أولًا عن عنوان الخادم المرتبط بـ eloquentJavaScript.net ويحاول فتح اتصال TCP معه على المنفَذ 80 الذي هو المنفَذ الافتراضي لحركة مرور HTTP، فإذا كان الخادم موجودًا ويقبل الاتصال فقد يرسل المتصفح شيئًا مثل هذا:

```
GET /18_http.html HTTP/1.1
Host: eloquentJavaScript.net
User-Agent: Your browser's name
```

ثم يستجيب الخادم من خلال نفس قناة الاتصال:

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

<!doctype html>
... the rest of the document
```

تكون أول كلمة هي التابع الخاص بالطلب، إذ تعني GET أننا نريد الحصول على مصدر بعينه، كما هناك توابع أخرى مثل DELETE لحذف المصدر وPUT لإنشائه أو استبداله وPOST لإرسال معلومات إليه. لاحظ أنّ الخادم ليس عليه تنفيذ جميع الطلبات التي تأتيه، فإذا ذهبتَ إلى موقع ما وطلبت حذف صفحته الرئيسية فسيرفض الخادم؛ أما الجزء الذي يلي اسم التابع، فيكون مسار المورد الذي يُطبق الطلب عليه، حيث يكون ملفًا على الخادم في أبسط حالاته، لكن البروتوكول لا يشترط كونه ملفًا فقط، بل قد يكون أي شيء يمكن نقله كما لو كان ملفًا، كما تولِّد العديد من الخوادم الاستجابات التي تنتجها لحظيًا، فإذا فتحت https://github.com/marijnh مثلًا، فسيبحث الخادم في قاعدة بياناته عن مستخدِم باسم marijnh، فإذا وجده فسيولِّد صفحة مستخدِم له.

يذكر أول سطر في الطلب بعد مسار المورد الـ HTTP/1.1 للإشارة إلى نسخة بروتوكول HTTP الذي يستخدِمه، كما تستخدِم مواقع كثيرة النسخة الثانية من HTTP عمليًا، إذ تدعم المفاهيم نفسها التي تدعمها النسخة الأولى 1.1، لكنها أعقد منها لتكون أسرع، كما ستبدِّل المتصفحات إلى البروتوكول المناسب تلقائيًا أثناء التحدث مع الخادم المعطى، وسيكون خرج الطلب هو نفسه بغض النظر عن النسخة المستخدَمة، لكننا سنركز على النسخة 1.1 بما أنها أبسط وأسهل في التعديل عليها. ستبدأ استجابة الخادم بالنسخة أيضًا تليها بحالة الاستجابة مثل شيفرة حالة من ثلاثة أرقام أولًا، ثم مثل سلسلة نصية مقروءة من قِبَل المستخدِم.

```
HTTP/1.1 200 OK
```

تبدأ رموز الحالة بـ 2 لتوضح نجاح الطلب؛ أما الطلبات التي تبدأ بـ 4 فتعني أنّ ثمة شيء خطأ في الطلب، ولعل أشهر رمز حالة HTTP هنا هي 404، والتي تعني أن المصدر غير موجود أو لا يمكن العثور عليه؛ أما الرموز التي تبدأ بالرقم 5، فتعني حدوث خطأ على الخادم ولا تتعلق المشكلة بالطلب نفسه، وقد يُتبع أول سطر من الطلب أو الاستجابة بعدد من الترويسات، وهي أسطر في صورة `name: value` توضِّح معلومات إضافية عن الطلب أو الاستجابة، وهي جزء من المثال الذي يوضح الاستجابة:

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Thu, 04 Jan 2018 14:05:30 GMT
```

يخبرنا هذا بحجم مستند الاستجابة ونوعه، وهو مستند HTML في هذه الحالة حجمه 65585 بايت، كما يخبرنا متى كانت آخر مرة عُدِّل فيها.

يملك كل من العميل والخادم في أغلب الترويسات حرية إدراجها في الطلب أو الاستجابة، لكن بعض الترويسات يكون إدراجها إلزاميًا مثل ترويسة `HOST` التي تحدد اسم المضيف hostname، إذ يجب إدراجها في الطلب لأن الخادم قد يخدِّم عدة أسماء مضيفين على عنوان IP واحد، فبدون الترويسة لن يعرف أيّ واحد فيها يقصده العميل الذي يحاول التواصل معه، وقد تدرِج الطلبات أو الاستجابات سطرًا فارغًا بعد اسم المضيف متبوعًا بمتن body يحتوي على البيانات المرسلة، كما لا ترسل طلبات `GET` و`DELETE` أيّ بيانات، على عكس طلبات `PUT` و`POST`، وبالمثل فقد لا تحتاج بعض أنواع الاستجابات إلى متن مثل استجابات الخطأ error responses.

18.2 المتصفحات وHTTP

رأينا في المثال السابق أنّ المتصفح سينشئ الطلب حين نكتب الرابط URL في شريط عنوانه، فإذا أشارت صفحة HTML الناتجة إلى ملفات أخرى مثل صور أو ملفات جافاسكربت، فسيجلب المتصفح هذه الملفات أيضًا، ومن المعتاد للمواقع متوسطة التعقيد إدراج من 10 إلى 200 مصدر مع الصفحة، كما سترسل المتصفحات عدة طلبات `GET` في الوقت نفسه بدلًا من انتظار الاستجابة الواحدة، ثم إرسال طلب آخر من أجل تحميل الصفحة بسرعة، وقد تحتوي صفحات HTML على استمارات forms تسمح للمستخدِم بملء بيانات وإرسالها إلى الخادم، وفيما يلي مثال عن استمارة:

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

تصف الشيفرة السابقة استمارةً لها حقلين أحدهما صغير يطلب الاسم والآخر أكبر لكتابة رسالة فيه، وتُرسَل الاستمارة عند الضغط على زر إرسال Send، أي يحزَّم محتوى حقلها في طلب HTTP وينتقل المتصفح إلى نتيجة ذلك الطلب.

تضاف المعلومات التي في الاستمارة إلى نهاية رابط `action` على أساس سلسلة استعلام نصية إذا كانت سمة العنصر `method` الخاص بالاستمارة `<form>` هي `GET` -أو إذا أُهملت-، وقد ينشئ المتصفح طلبًا إلى هذا الرابط:

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

تحدِّد علامة الاستفهام نهاية جزء المسار من الرابط وبداية الاستعلام، كما تُتبع بأزواج من الأسماء والقيم تتوافق مع سمة name في عناصر حقول الاستمارة ومحتوى تلك العناصر على الترتيب، ويُستخدَم محرف الإضافة ampersand أي & لفصل تلك الأزواج، في حين تكون الرسالة الفعلية المرمَّزة في الرابط هي "Yes?"، لكن ستُستبدَل شيفرة غريبة بعلامة الاستفهام، كما يجب تهريب بعض المحارف في سلاسل الاستعلامات النصية، فعلامة الاستفهام الممثلة بـ %3F هي أحد تلك المحارف، وسنجد أن هناك شبه قاعدة غير مكتوبة تقول أنّ كل صيغة تحتاج إلى طريقتها الخاصة في تهريب المحارف، وهذه التي بين أيدينا تُسمى ترميز الروابط التشعبية URL encoding، حيث تستخدِم علامة النسبة المئوية ويليها رقمين ست-عشريين يرمِّزان شيفرة المحرف، وفي حالتنا تكون 3F -التي هي 63 في النظام العشري- شيفرة محرف علامة الاستفهام، وتوفِّر جافاسكربت الدالتين encodeURIComponent وdecodeURIComponent من أجل ترميز تلك الصيغة وفك ترميزها أيضًا.

```
console.log(encodeURIComponent("Yes?"));
// → Yes%3F
console.log(decodeURIComponent("Yes%3F"));
// → Yes?
```

إذا غيرنا السمة method لاستمارة HTML في المثال الذي رأيناه إلى POST، فسيستخدِم طلب HTTP الذي أنشئ لإرسال الاستمارة التابع POST ويضع سلسلة الاستعلام النصية في متن الطلب بدلًا من إضافتها إلى الرابط.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

يجب استخدام طلبات GET للطلبات التي تطلب معلومات فقط وليس لها تأثيرات جانبية؛ أما الطلبات التي تغيِّر شيئًا في الخادم مثل إنشاء حساب جديد أو نشر رسالة، فيجب التعبير عنها بتوابع أخرى مثل POST، كما تدرك برامج العميل مثل المتصفحات أنها يجب ألا تنشئ طلبات POST عشوائيًا وإنما تنشئ طلبات GET أولًا ضمنيًا لجلب المصدر التي تظن أنّ المستخدِم سيحتاج إليه قريبًا، كما سنعود إلى كيفية التفاعل مع الاستمارات من جافاسكربت لاحقًا في هذا الفصل.

18.3 واجهة Fetch

تُسمى الواجهة التي تستطيع جافاسكربت الخاصة بالمتصفح إنشاء طلبات HTTP من خلالها باسم fetch، وبما أنها جديدة نسبيًا فستستخدم الوعود promises وهو الأمر النادر بالنسبة لواجهات المتصفحات.

```
fetch("example/data.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

يعيد استدعاء `fetch` وعدًا يُحل إلى كائن `Response` حاملًا معلومات عن استجابة الخادم مثل شيفرة حالته وترويساته، وتغلَّف الترويسات في كائن شبيه بالخارطة `Map`-like الذي يهمل حالة الأحرف في مفاتيحه أي أسماء الترويسات- لأنه لا يفترض أن تكون أسماء الترويسات حساسةً لحالة الأحرف. هذا يعني أن كلا مما يلي:

```
headers.get("Content-Type")
headers.get("content-TYPE")
```

سيُعيدان القيمة نفسها.

لاحظ أن الوعد الذي تعيده `fetch` يُحل بنجاح حتى لو استجاب الخادم برمز خطأ، وقد يُرفض إذا كان ثمة خطأ في الشبكة أو لم يوجد الخادم الذي أرسل إليه الطلب، كما يجب أن يكون أول وسيط لواجهة `fetch` هو الرابط الذي يراد طلبه، فإذا لم يبدأ الرابط باسم البروتوكول http: مثلًا فسيعامَل نسبيًا، أي يفسَّر وفق المستند الحالي، وإذا بدأ بشرطة مائلة `/` فسيستبدل المسار الحالي الذي يكون جزء المسار الذي يلي اسم الخادم، أما إذا لم يبدأ بالشرطة المائلة، فسيوضع جزء المسار الحالي إلى آخر محرف شرطة مائلة -مع الشرطة نفسها- أمام الرابط النسبي.

يُستخدَم التابع `text` للحصول على المحتوى الفعلي للاستجابة، ويعيد هذا التابع وعدًا لأن الوعد الأولي يُحَل عند استقبال ترويسات الاستجابة، ولأنّ قراءة متن الاستجابة قد تستغرق وقتًا أطول.

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
// → This is the content of data.txt
```

يعيد التابع `json` -وهو تابع شبيه بالسابق- وعدًا يُحل إلى القيمة التي تحصل عليها حين تحلل المتن مثل JSON أو يُرفض إذا لم يكن JSON صالحًا، كما تستخدِم واجهة `fetch` التابع `GET` افتراضيًا لإنشاء طلبها ولا تدرِج متن الطلب، كما يمكنك إعدادها لغير ذلك بتمرير كائن له خيارات إضافية على أساس وسيط ثاني، فهذا الطلب مثلًا يحاول حذف `example/data.txt`:

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

يعني رمز الحالة 405 أنّ "الطلب غير مسموح به" وهو أسلوب خادم HTTP ليقول "لا أستطيع فعل هذا"، كما يمكن إضافة الخيار body لإضافة متن الطلب، كما يُستخدَم الخيار headers لضبط الترويسات، فهذا الطلب مثلًا يضمِّن الترويسة Range التي تخبر الخادم بإعادة جزء من الاستجابة فقط.

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → المحتوى
```

سيضيف المتصفح بعض ترويسات الطلب تلقائيًا مثل Host وتلك المطلوبة كي يعرف الخادم حجم المتن، لكن ستكون إضافة ترويساتك الخاصة مفيدةً إذا أردنا إضافة أشياء مثل معلومات التوثيق، أو لنخبر الخادم بصيغة الملف التي نريد استقبالها.

18.4 صندوق اختبارات HTTP

لا شك أنّ إنشاء طلبات HTTP في سكربتات صفحة الويب سيرفع علامات استفهام حول الأمان، فالشخص الذي يتحكم بالسكربت قد لا تكون لديه دوافع الشخص نفسها التي يشغلها على حاسوبه، فإذا زرنا الموقع themafia.org مثلًا، فلا نريد لسكربتاته أن تكون قادرةً على إنشاء طلب إلى mybank.com باستخدام معلومات التعريف من متصفحنا مع تعليمات بتحويل جميع أموالنا إلى حساب عشوائي، لهذا تحمينا المتصفحات من خلال عدم السماح للسكربتات بإنشاء طلبات HTTP إلى نطاقات أخرى (أسماء نطاقات مثل themafia.org وmybank.com)، وتُعَدّ هذه مشكلةً مؤرقةً عند بناء أنظمة تريد الوصول إلى عدة نطاقات من أجل أسباب مشروعة ومنطقية، ولحسن الحظ تستطيع الخوادم إدراج ترويسة لهذا الغرض في استجابتها لإخبار المتصفح صراحةً أن هذا الطلب يمكن أن يأتي من نطاق آخر:

```
Access-Control-Allow-Origin: *
```

18.5 تقدير HTTP

هناك عدة طرق مختلفة لنمذجة التواصل بين برامج جافاسكربت العاملة في المتصفح -جانب العميل- والبرنامج الذي على الخادم -أي جانب الخادم-، وإحدى أكثر تلك الطرق استخدامًا هي استدعاءات الإجراءات البعيدة remote procedure calls، إذ يتبع التواصل في هذا النموذج أنماط استدعاءات الدوال العادية عدا أنّ

الدالة تعمل فعليًا على حاسوب آخر، حيث يتطلب استدعاؤها إنشاء طلب إلى الخادم الذي يتضمن اسم الدالة والوسائط، كما تحتوي استجابة ذلك الطلب على القيمة المعادة.

عند التفكير في شأن استدعاءات الإجراء البعيد، لا يكون HTTP أكثر من وسيلة تواصل، وستكتب على الأرجح طبقةً مجردةً تخفيه كليًا، كما يوجد هناك منظور آخر نبني فيه التواصل حول مفهوم الموارد وتوابع HTTP، فبدلًا من `addUser` المستدعى استدعاءً بعيدًا، فإننا فسنرسل طلبية `PUT` إلى `/users/larry`، وبدلًا من تمرير خصائص ذلك المستخدِم في وسائط الدالة، فإنك تعرِّف صيغة مستند JSON من أجل تمثيل المستخدِم أو تستخدم صيغةً موجودةً من قبل لذلك، ويمثل آنذاك جسم طلبية `PUT` المرسلة -أي الكائن `body` المرسل في الطلبية- لإنشاء مورد جديد مستندًا يخص ذلك المورد المراد إنشاؤه وتخزينه.

كما يُجلَب المورد بإنشاء طلب `GET` إلى رابط المورد -`/users/larry` مثلًا- والذي يُعيد المستند الممثل للمورد، إذ يسهل هذا المنظور استخدام بعض المزايا التي يوفرها HTTP مثل دعم تخزين الموارد -أي إنشاء نسخة مؤقتة عند العميل من أجل تسريع الوصول-، كما توفر المفاهيم المستخدَمة في HTTP مجموعة مبادئ مفيدة في تصميم واجهة الخوادم الخاصة بك بما أنها جيدة التصميم.

18.6 الأمان وHTTP

تمر البيانات المتنقلة عبر الإنترنت في طرق محفوفة بالمخاطر، إذ تمر خلال أي طريق نتواجد فيه سواء كان نقطة اتصال في مقهى أو شبكات تتحكم بها شركات أو دول بغية الحصول على وجهتها، وقد تُعتَرض وتفتَّش في أي نقطة في طريقها وقد تُعدَّل أيضًا، وهنا لا يكفي بروتوكول HTTP العادي بما أنّ بعض البيانات سرية مثل كلمات مرور بريدك أو يجب عليها الوصول إلى وجهتها دون تعديل مثل رقم الحساب الذي تحوِّل المال إليه من خلال موقع البنك الذي تتعامل معه.

نستخدِم هنا بروتوكولًا أحدث هو HTTPS الذي نجده في الروابط التي تبدأ بـ https://، إذ يغلِّف حركة مرور HTTP بطريقة تصعب قراءتها والتعديل عليها، ويؤكد الطرف العميل قبل إرسال البيانات أنّ الخادم الذي يطلبها هو نفسه وليس منتحلًا له من خلال التأكد من شهادة مشفرة مصدَرة من جهة توثيق يعتمدها المتصفح، ثم تشفَّر جميع البيانات بطريقة تمنع استراق النظر إليها أو التعديل عليها، وعليه يمنع HTTPS أيّ جهة خارجية من انتحال الموقع الذي تريد التواصل معه ومن اختلاس النظر أو التجسس على تواصلكما، لكنه ليس مثاليًا بالطبع فقد وقعت عدة حوادث فشل فيها HTTPS بسبب شهادات مزورة أو مسروقة وبسبب برامج مخترَقة أو معطوبة، لكنه أكثر أمانًا من HTTP العادي.

18.7 حقول الاستمارات

صُمِّمت الاستمارات ابتداءً للويب قبل مجيء جافاسكربت من أجل السماح لمواقع الويب إرسال البيانات التي يدخلها المستخدِم في هيئة طلب HTTP، حيث يفترِض هذا التصميم أنّ التفاعل مع الخادم سيحدث دائمًا

من خلال الانتقال إلى صفحة جديدة، غير أنّ عناصرها جزء من نموذج كائن مستند DOM مثل بقية الصفحة، كما تدعم عناصر DOM التي تمثِّل حقول الاستمارة عددًا من الخصائص والأحداث التي ليست موجودةً في العناصر الأخرى، حيث تمكننا من فحص حقول الإدخال تلك والتحكم فيها ببرامج جافاسكربت وأمور أخرى مثل إضافة وظيفة جديدة إلى استمارة أو استخدام استمارات وحقول على أساس وحدات بناء في تطبيق جافاسكربت.

تتكون استمارة الويب من عدد من حقول الإدخال تُجمع في وسم `<form>`، وتسمح HTML بعدة تنسيقات من الحقول بدايةً من أزرار الاختيار checkboxes إلى القوائم المنسدلة وحقول إدخال النصوص، ولن نناقش كل أنواع الحقول في هذا الكتاب لكن سنبدأ بنظرة عامة عليها.

تستخدِم أكثر أنواع الحقول وسم `<input>` وتُستخدَم السمة `type` الخاصة بهذا الوسم من أجل اختيار تنسيق الحقل، وفيما يلي أكثر أنواع `<input>` المستخدَمة:

text	حقل نصي ذو سطر واحد
`password`	حقل نصي مثل `text` لكن يخفي النص الذي يُكتَب فيه
`checkbox`	مفتاح تشغيل/إغلاق
`radio`	جزء من حقل اختيار من متعدد
`file`	يسمح للمستخدِم اختيار ملف من حاسوبه

يمكن وضع حقول الاستمارات في أي مكان في الصفحة ولا يشترط ظهورها في وسوم `<form>` وحدها، كما لا يمكن إرسال تلك الحقول التي تكون مستقلة بذاتها وخارج استمارة، فالاستمارات وحدها هي التي ترسَل، لكن على أيّ حال لا نحتاج إلى إرسال محتوى حقولنا بالطريقة التقليدية عند استخدام جافاسكربت في الاستجابة للمدخلات.

```
<p><input type="text" value="abc"> (text)</p>
<p><input type="password" value="abc"> (password)</p>
<p><input type="checkbox" checked> (checkbox)</p>
<p><input type="radio" value="A" name="choice">
   <input type="radio" value="B" name="choice" checked>
   <input type="radio" value="C" name="choice"> (radio)</p>
<p><input type="file"> (file)</p>
```

تختلف واجهة جافاسكربت لمثل تلك العناصر باختلاف نوع العنصر.

تمتلك الحقول النصية متعددة الأسطر وسمًا خاصًا بها هو `<textarea>`، وذلك بسبب غرابة استخدام سمة لتحديد قيمة ابتدائية لسطر متعدد، كما يجب إغلاق هذا الوسم بأسلوب الإغلاق المعتاد في HTML

بإضافة `</textarea>`، حيث يُستخدَم النص الموجود بين هذين الوسمين على أساس نص ابتدائي بدلًا من سمة `value`.

```
<textarea>
one
two
three
</textarea>
```

أخيرًا، يُستخدَم الوسم `<select>` لإنشاء حقل يسمح للمستخدِم بالاختيار من عدد من الخيارات المعرَّفة مسبقًا، ويُطلَق الحدث `"change"` كلما تغيرت قيمة حقل من حقول الاستمارة.

```
<select>
  <option>Pancakes</option>
  <option>Pudding</option>
  <option>Ice cream</option>
</select>
```

18.8 التركيز Focus

تستطيع حقول الاستمارات الحصول على نشاط لوحة المفاتيح عند النقر أو تفعيلها بأيّ شكل آخر على عكس أغلب عناصر مستندات HTML، بحيث تصبح هي العنصر المفعَّل الحالي ومستقبل إدخال لوحة المفاتيح، وعلى ذلك نستطيع الكتابة في الحقل النصي حين يكون مركَّزًا فقط؛ أما الحقول الأخرى فتختلف في استجابتها لأحداث لوحة المفاتيح، إذ تحاول قائمة `<select>` مثلًا الانتقال إلى الخيار الذي يحتوي النص الذي كتبه المستخدِم وتستجيب لمفاتيح الأسهم عبر تحريك اختيارها لأعلى وأسفل.

يمكننا التحكم في التركيز باستخدام جافاسكربت من خلال التابعَين `focus` و`blur`، حيث ينقل `focus` التركيز إلى عنصر DOM الذي استدعي عليه؛ أما الثاني فسيزيل التركيز منه، وتتوافق القيمة التي في `document.activeElement` مع العنصر المركَّز حاليًا.

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
```

```
  // → BODY
</script>
```

يُتوقَّع من المستخدِم في بعض الصفحات أن يرغب في التفاعل مع أحد حقول الاستمارة فورًا، ويمكن استخدام جافاسكربت لتركيز ذلك الحقل عند تحميل المستند، لكن توفر HTML السمة `autofocus` أيضًا، والتي تعطينا التأثير نفسه وتخبر المتصفح بما نحاول فعله، وهذا يعطي المتصفح خيار تعطيل السلوك إذا كان غير مناسب كما في حالة محاولة المستخدِم تركيز حقل آخر أو عنصر آخر، وقد تعارفت المتصفحات على تمكين المستخدم من نقل التركيز خلال المستند بمجرد ضغط زر جدول أو tab على لوحة المفاتيح، ونستطيع هنا التحكم في الترتيب الذي تستقبل به العناصر ذلك التركيز باستخدام السمة `tabindex`، وسيجعل المثال التالي التركيز يقفز من المدخلات النصية إلى زر OK بدلًا من المرور على رابط المساعدة أولًا:

```
<input type="text" tabindex=1> <a href=".">(help)</a>
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

السلوك الافتراضي لأغلب عناصر HTML أنها لا يمكن تركيزها، غير أنك تستطيع إضافة سمة `tabindex` إلى أي عنصر لجعله قابلًا للتركيز؛ أما إذا جعلنا قيمتها `-1` فسيتم تخطي العنصر حتى لو كان قابلًا للتركيز.

18.9 الحقول المعطلة

يمكن تعطيل أيّ حقل من حقول الاستمارات من خلال السمة `disabled` الخاصة بها، وهي سمة يمكن تحديدها دون قيمة، إذ يعطِّل وجودها الحقل مباشرةً، ولا يمكن تركيز الحقول المعطَّلة أو تغييرها، كما ستظهرها المتصفحات بلون رمادي وباهت.

```
<button>أنا مركَّز الآن</button>
<button disabled>خرجت!</button>
```

إذا عالج البرنامج إجراءً سببه زر أو تحكم آخر قد يحتاج إلى تواصل مع الخادم وسيستغرق وقتًا، فمن الأفضل تعطيل التحكم حتى انتهاء ذلك الإجراء، وهكذا لن يتكرر الإجراء إذا نفذ صبر المستخدِم ونقر عليه مرةً أخرى.

18.10 الاستمارات على أساس عنصر كامل

إذا احتوى الحقل على العنصر `<form>` فسيحتوي عنصر DOM الخاص به على الخاصية `form` التي تربطه إلى عنصر DOM الخاص بالاستمارة، ويحتوي العنصر `<form>` بدوره على خاصية تسمى `elements` تحوي تجميعةً شبيهةً بالمصفوفة من الحقول التي بداخلها. كذلك تحدِّد السمة `name` الموجودة في حقل الاستمارة الطريقة التي تعرَّف بها قيمتها عند إرسال الاستمارة، كما يمكن استخدامها على أساس اسم خاصية عند الوصول إلى الخاصية `elements` الخاصة بالاستمارة والتي تتصرف على أساس كائن شبيه بالمصفوفة -يمكن الوصول إليه بعدد-، وخريطة map -يمكن الوصول إليها باسم-.

```
<form action="example/submit.html">
  Name: <input type="text" name="name"><br>
  Password: <input type="password" name="password"><br>
  <button type="submit">Log in</button>
</form>
<script>
  let form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

يرسل الزر الذي فيه سمة `type` الخاصة بـ `submit` الاستمارة عند الضغط عليه، كما سنحصل على التأثير نفسه عند الضغط على زر الإدخال Enter وإذا كان حقل الاستمارة مركَّزًا، ويعني إرسال الاستمارة غالبًا أنّ المتصفح ينتقل إلى الصفحة التي تحددها سمة `action` الخاصة بالاستمارة مستخدِمًا أحد الطلبَين `GET` أو `POST`، لكن يُطلَق الحدث `"submit"` قبل حدوث ذلك، وتستطيع معالجة هذا الحدث بجافاسكربت، ونمنع ذلك السلوك الافتراضي من خلال استدعاء `preventDefault` على كائن الحدث.

```
<form action="example/submit.html">
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Saving value", form.elements.value.value);
    event.preventDefault();
  });
</script>
```

يملك اعتراض أحداث `"submit"` في جافاسكربت فوائد عديدة، حيث نستطيع كتابة شيفرة للتحقق من أنّ القيم التي أدخلها المستخدِم منطقية، ونخرِج رسائل خطأ له إذا وجدنا أخطاءً في تلك القيم بدلًا من إرسال

الاستمارة، أو نستطيع تعطيل الطريقة المعتادة في إرسال الاستمارة بالكامل كما في المثال، ونجعل البرنامج يعالِج المدخلات باستخدام `fetch` لإرسالها إلى خادم دون إعادة تحميل الصفحة.

18.11 الحقول النصية

تشترك الحقول التي أنشئت بواسطة الوسوم `<textarea>` أو `<input>` مع النوع `text` و`password` واجهةً مشتركةً، كما تحتوي عناصر DOM الخاصة بها على الخاصية `value` التي تحمل محتواها الحالي على أساس قيمة نصية، وإذا عيّنا تلك الخاصية إلى سلسلة نصية أخرى فسيتغيَّر محتوى الحقل.

تعطينا كذلك الخصائص `selectionStart` و`selectionEnd` الخاصة بالحقول النصية معلومات عن المؤشر والجزء المحدَّد في النص، فإذا لم يكن ثمة نص محدَّد، فستحمل هاتان الخاصيتان العدد نفسه والذي يشير إلى موضع المؤشر، حيث يشير 0 مثلًا إلى بداية النص، ويشير 10 إلى أنّ المؤشر بعد المحرف العاشر؛ أما إذا حدِّد جزء من الحقل، فستختلف هاتين الخاصيتين لتعطينا بداية النص المحدَّد ونهايته، كما يمكن الكتابة فيهما مثل `value` تمامًا.

لنفترض أنك تكتب مقالةً عن الفرعون "خع سخموي Khasekhemwy" لكن لا نستطيع تهجئة اسمه، لذا تساعدنا هنا الشيفرة التالية التي توصل وسم `<textarea>` بمعالج حدث يُدخل النص Khasekhemwy لك إذا ضغطت على مفتاح F2.

```
<textarea></textarea>
<script>
  let textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", event => {
    // The key code for F2 happens to be 113
    if (event.keyCode == 113) {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    let from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
                  field.value.slice(to);
    // ضع المؤشر بعد الكلمة
    field.selectionStart = from + word.length;
    field.selectionEnd = from + word.length;
```

```
  }
</script>
```

تستبدِل الدالة `replaceSelection` الكلمة الصعبة المعطاة والتي نريدها بالجزء المحدَّد حاليًا من محتوى الحقل النصي، ثم تنقل المؤشر بعد تلك الكلمة كي يستطيع المستخدِم متابعة الكتابة، ولا يُطلَق الحدث `"change"` للحقل النصي في كل مرة يُكتب شيء ما، بل عندما يزال التركز عن أحد الحقول بعد تغيُّر محتواه فقط، ويجب علينا تسجيل معالج للحدث `"input"` من أجل الاستجابة الفورية للتغيرات الحادثة في الحقل النصي، حيث يُطلَق في كل مرة يكتب المستخدِم فيها محرفًا أو يحذف نصًا أو يعدِّل في محتوى الحقل، ويظهر المثال التالي حقلًا نصيًا وعدّادًا يعرض الطول الحالي للنص في الحقل:

```
<input type="text"> length: <span id="length">0</span>
<script>
  let text = document.querySelector("input");
  let output = document.querySelector("#length");
  text.addEventListener("input", () => {
    output.textContent = text.value.length;
  });
</script>
```

18.12 أزرار الاختيار وأزرار الانتقاء

يُعَدّ حقل زر الاختيار checkbox مخيّرًا فهو يخير بين اختياره من المجموعة أو لا ويمكن استخراج قيمته أو تغييرها من خلال الخاصية `checked` الخاصة به والتي تحمل قيمة بوليانية.

```
<label>
  <input type="checkbox" id="purple"> Make this page purple
</label>
<script>
  let checkbox = document.querySelector("#purple");
  checkbox.addEventListener("change", () => {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

يربط وسم العنوان `<label>` جزءًا من المستند بحقل إدخال، فإذا نقرنا في أيّ مكان على العنوان فسيُركز على حقل الإدخال وتغيَّر قيمته (بتحديده أو إزالة التحديد منه بحسب نوع الزر).

يشبه زر الانتقاء زر الاختيار لكنه يرتبط ضمنيًا بأزرار انتقاء أخرى لها سمة name نفسها كي يُنتقى واحد منها فقط، وقد سميت بهذا الاسم لأنها تشبه أزرار اختيار محطات الراديو في أجهزة المذياع في شكلها ووظيفتها، حيث إذا ضغطنا على أحد تلك الأزرار فإن بقية الأزرار تقفز إلى الخارج ولا تعمل سوى المحطة صاحبة الزر المنضغط.

```
Color:
<label>
  <input type="radio" name="color" value="orange"> Orange
</label>
<label>
  <input type="radio" name="color" value="lightgreen"> Green
</label>
<label>
  <input type="radio" name="color" value="lightblue"> Blue
</label>
<script>
  let buttons = document.querySelectorAll("[name=color]");
  for (let button of Array.from(buttons)) {
    button.addEventListener("change", () => {
      document.body.style.background = button.value;
    });
  }
</script>
```

تُستخدَم الأقواس المربعة في استعلام CSS المعطى إلى querySelectorAll لمطابقة السمات، وهي تختار العناصر التي تكون سمة name الخاصة بها هي "color".

18.13 حقول التحديد

تسمح حقول التحديد select fields للمستخدِم الاختيار من بين مجموعة خيارات كما في حالة أزرار الانتقاء، لكن مظهر الوسم <select> يختلف بحسب المتصفح على عكس أزرار الانتقاء التي نتحكم في مظهرها، كما تحتوي هذه الحقول على متغير يشبه قائمة أزرار الاختيار أكثر من أزرار الانتقاء، فإذا أعطينا وسم <select> السمة multiple، فسيسمح للمستخدِم اختيار أيّ عدد من الخيارات التي يريدها بدلًا من خيار واحد، وسيكون مظهر هذا مختلفًا باختلاف المتصفح، لكنه سيختلف عن حقل التحديد العادي الذي يكون تحكمًا منسدلًا drop-down control لا يعرض الخيارات إلا عند فتحه.

يحتوي كل وسم `<option>` على قيمة يمكن تعريفها باستخدام السمة `value`، وإذا لم تعطى تلك القيمة، فسيُعَدّ النص الذي بداخل الخيار هو قيمته، كما تعكس خاصية `value` الخاصة بالعنصر `<select>` الخيار المحدَّد حاليًا، لكن لن تكون هذه الخاصية ذات شأن في حالة الحقل `multiple` بما أنها ستعطي قيمة خيار واحد فقط من الخيارات المحدَّدة الحالية، ويمكن الوصول إلى وسوم `<option>` الخاصة بحقل `<select>` على أساس كائن شبيه بالمصفوفة من خلال خاصية الحقل `options`، كما يحتوي كل خيار على خاصية تسمى `selected` توضِّح هل الخيار محدَّد حاليًا أم لا، ويمكن كتابة الخاصية لتحديد خيار ما أو إلغاء تحديده.

يستخرِج المثال التالي القيم المحدَّدة من حقل التحديد `multiple` ويستخدِمها لتركيب عدد ثنائي من بِتّات منفصلة، لتحديد عدة خيارات اضغط باستمرار على زر control أو command على ماك Mac.

```
<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  let select = document.querySelector("select");
  let output = document.querySelector("#output");
  select.addEventListener("change", () => {
    let number = 0;
    for (let option of Array.from(select.options)) {
      if (option.selected) {
        number += Number(option.value);
      }
    }
    output.textContent = number;
  });
</script>
```

18.14 حقول الملفات

صُمِّمت حقول الملفات ابتداءً على أساس طريقة لرفع الملفات من حاسوب المستخدِم من خلال استمارة؛ أما في المتصفحات الحديثة، فهي توفر طريقةً لقراءة تلك الملفات، ولكن من خلال برامج جافاسكربت، إذ يتصرف الحقل على أساس حارس لبوابة، بحيث لا تستطيع السكربت البدء بقراءة ملفات خاصة بالمستخدِم من حاسوبه، لكن إذا اختار المستخدِم ملفًا في حقل كهذا، فسيفسِّر المتصفح ذلك الإجراء على أنه سماح للسكربت

بقراءة الملف، ويبدو حقل الملف على أساس زر عليه عنوان مثل "اختر الملف" أو "تصفح الملف" مع معلومات عن الملف المختار تظهر إلى جانبه.

```
<input type="file">
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0];
      console.log("You chose", file.name);
      if (file.type) console.log("It has type", file.type);
    }
  });
</script>
```

تحتوي الخاصية `files` لعنصر حقل الملف على الملفات المختارة في الحقل، وهي كائن شبيه بالمصفوفة وليست مصفوفةً حقيقيةً، كما تكون فارغةً في البداية، والسبب في عدم وجود خاصية مستقلة باسم `file`، هو دعم الحقول لسمة `multiple` التي تجعل من الممكن تحديد عدة ملفات في الوقت نفسه، كما تحتوي الكائنات في كائن `files` على خاصيات مثل `name` لاسم الملف و`size` لحجمه مقدَّرًا بالبايت -الذي هو وحدة قياس تخزينية تتكون من 8 بِتّات-، كما تحتوي على الخاصية `type` التي تمثِّل نوع وسائط media الملف التي قد تكون نصًا عاديًا `text/plain` أو صورةً `image/jpeg`، لكن ليس لتلك الكائنات خاصيةً يكون فيها محتوى الملف، وبما أنّ قراءة الملف من القرص ستستغرق وقتًا، فيجب أن تكون الواجهة غير تزامنية لتجنب تجميد أو تعليق المستند أثناء قراءته.

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
```

```
  });
</script>
```

تتم عملية قراءة الملف من خلال إنشاء كائن `FileReader` الذي يسجِّل معالج الحدث `"load"` له، ويستدعي التابع `readAsText` الخاص به ليعطيه الملف الذي نريد قراءته، كما ستحتوي الخاصية `result` الخاصة بالقارئ على محتويات الملف بمجرد انتهاء التحميل، ويطلق الكائن `FileReader` أيضًا حدث خطأ `"error"` عند فشل قراءة الملف لأيّ سبب، إذ سيؤول كائن الخطأ نفسه في خاصية `error` الخاصة بالقارئ، ورغم تصميم تلك الواجهة قبل أن تصبح الوعود promises جزءًا من جافاسكربت، إلا أنك تستطيع تغليفها بوعد كما يلي:

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}
```

18.15 تخزين البيانات في جانب العميل

ستكون صفحات HTML البسيطة التي فيها قليل من جافاسكربت صيغةً رائعةً من أجل التطبيقات المصغَّرة، وهي برامج مساعِدة صغيرة تؤتمت مهامًا أساسية عبر توصيل بعض حقول الاستمارات بمعالِجات الأحداث، حيث يمكنك فعل أيّ شيء بدءًا من التحويل بين وحدات القياس المختلفة إلى حساب كلمات المرور من كلمة مرور رئيسية واسم موقع.

لكن لا نستطيع استخدام رابطات جافاسكربت إذا احتاج مثل ذلك التطبيق إلى تذكر أمر بين جلساته sessions، ذلك أنّ هذه الرابطات تُحذَف عند كل إغلاق للصفحة، غير أنه يمكن إعداد خادم وتوصيله بالإنترنت وجعل التطبيق يخزن هناك، وسنرى كيفية فعل ذلك في الفصل العشرين، لكن المقام يقصر هنا عن شرحه لتعقيده، وعلى أيّ حال، من المفيد أحيانًا إبقاء البيانات في المتصفح، كما يمكن استخدام الكائن `localStorage` لتخزين البيانات بطريقة تبقيها موجودةً حتى مع إعادة تحميل الصفحات، حيث يسمح لك ذلك الكائن بتصنيف قيم السلاسل النصية تحت أسماء.

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

تبقى القيمة الموجودة في `localStorage` إلى أن يُكتَب فوقها، كما تُحذَف باستخدام `removeItem` أو بمحو المستخدِم لبياناته المحلية، وتحصل المواقع التي هي من نطاقات مختلفة على أقسام تخزين مختلفة، ويعني هذا نظريًا عدم إمكانية قراءة وتعديل البيانات المخزَّنة في `localStorage` من قِبل موقع ما إلا بسكربتات من نفس الموقع، كما تضع المتصفحات حدًا لحجم البيانات التي يمكن للمتصفح أن يخزِّنها أي `localStorage`، وذلك لمنع تلك الخاصية من ملء أقراص المستخدِمين الصلبة ببيانات عديمة الفائدة واستخدام مساحات كبيرة بها، وتنفِّذ الشيفرة التالية تطبيقًا لكتابة الملاحظات، حيث يحتفظ بمجموعة من الملاحظات المسماة ويسمح للمستخدِم بتعديل الملاحظات وإنشاء الجديد منها.

```
Notes: <select></select> <button>Add</button><br>
<textarea style="width: 100%"></textarea>

<script>
  let list = document.querySelector("select");
  let note = document.querySelector("textarea");

  let state;
  function setState(newState) {
    list.textContent = "";
    for (let name of Object.keys(newState.notes)) {
      let option = document.createElement("option");
      option.textContent = name;
      if (newState.selected == name) option.selected = true;
      list.appendChild(option);
    }
    note.value = newState.notes[newState.selected];

    localStorage.setItem("Notes", JSON.stringify(newState));
    state = newState;
  }
  setState(JSON.parse(localStorage.getItem("Notes")) || {
    notes: {"shopping list": "Carrots\nRaisins"},
```

```
      selected: "shopping list"
    });

    list.addEventListener("change", () => {
      setState({notes: state.notes, selected: list.value});
    });
    note.addEventListener("change", () => {
      setState({
        notes: Object.assign({}, state.notes,
                             {[state.selected]: note.value}),
        selected: state.selected
      });
    });
    document.querySelector("button")
      .addEventListener("click", () => {
        let name = prompt("Note name");
        if (name) setState({
          notes: Object.assign({}, state.notes, {[name]: ""}),
          selected: name
        });
      });
  </script>
```

تحصل هذه السكربت على حالتها من القيمة `"Notes"` المخزَّنة في `localStorage`، وإذا لم تكن موجودة، فستنشئ حالة مثال وليس فيها إلا قائمة تسوق، ونحصل على القيمة `null` عند محاولة قراءة حقل غير موجود من `localStorage`، كما أنّ تمرير `null` إلى `JSON.parse` سيجعله يحلل السلسلة النصية `"null"` ويُعيد `null`، وعلى ذلك يمكن استخدام العامِل `||` لتوفير قيمة افتراضية في مثل هذه المواقف؛ أما التابع `setState` فيتأكد أنّ DOM يُظهِر حالةً معطاةً ويخزِّن الحالة الجديدة في `localStorage`، كما تستدعي معالِجات الأحداث هذه الدالة للانتقال إلى حالة جديدة.

كان استخدام `Object.assign` في المثال السابق من أجل إنشاء كائن جديد يكون نسخة من `state.notes` القديم، لكن مع خاصية واحدة مضافة أو مكتوب فوقها، وتأخذ `Object.assign` وسيطها الأول وتضيف جميع الخاصيات من أيّ وسائط آخرين إليه، فإذا أعطيناها كائنًا فارغًا فسنجعلها تملأ كائنًا جديدًا، وتُستخدَم الأقواس المربعة في الوسيط الثالث لإنشاء خاصية اسمها يكون مبنيًا على قيمة ديناميكية، كما لدينا كائن آخر يشبه `localStorage` اسمه `sessionStorage`، والاختلاف بين الاثنين هو أنّ محتوى الأخير يُنسى بنهاية كل جلسة، حيث يحدث عند إغلاق المتصفح وذلك بالنسبة لأغلب المتصفحات.

18.16 خاتمة

ناقشنا في هذا الفصل كيفية عمل بروتوكول HTTP، وقلنا أنّ العميل يرسل الطلب الذي يحتوي على تابع يكون GET في الغالب ومسار يحدِّد المصدر، ثم يقرر الخادم بعدها ماذا يفعل بذلك الطلب ويستجيب بشيفرة حالة ومتن استجابة، وقد يحتوي كل من الطلب والاستجابة على ترويسات توفر لنا معلومات إضافية، كما تُسمى الواجهة التي تستطيع جافاسكربت التي في المتصفح إنشاء طلبات HTTP منها باسم fetch، إذ يبدو إنشاء الطلب كما يلي:

```
fetch("/18_http.html").then(r => r.text()).then(text => {
  console.log(`The page starts with ${text.slice(0, 15)}`);
});
```

كما عرفنا من قبل، تنشئ المتصفحات طلبات GET لجلب الموارد المطلوبة لعرض صفحة ويب، وقد تحتوي الصفحة على استمارات تسمح للمستخدِم بإدخال المعلومات التي سترسلها بعدها في صورة طلب إلى الصفحة الجديدة عند إرسال الاستمارة، كما تستطيع HTML تمثيل عدة أنواع من حقول الاستمارات مثل الحقول النصية وأزرار الاختيار وحقول الاختيار من متعدد ومختارات الملفات file pickers، إذ يمكن فحص مثل تلك الحقول وتعديلها باستخدام جافاسكربت، وهي تطلق الحدث "change" عند تعديلها والحدث "input" عند كتابة نص فيها، كما تستقبل أحداث لوحة المفاتيح عند انتقال نشاط لوحة المفاتيح إليها.

عرفنا أيضًا أنّ الخاصيات مثل value -المستخدَمة في النصوص وحقول التحديد- أو checked - المستخدَمة في أزرار الاختيار وأزرار الانتقاء-، تُستخدَم من أجل قراءة أو تعيين محتوى الحقل، كما رأينا أن الحدث "submit" يُطلَق عند إرسال الاستمارة عليها، ويستطيع معالِج جافاسكربت استدعاء preventDefault على ذلك الحدث من أجل تعطيل السلوك الافتراضي للمتصفح، كما قد توجد عناصر حقول الاستمارات خارج وسم الاستمارة نفسه.

إذا اختار المستخدم ملفًا من نظام ملفاته المحلي في حقل مختار الملفات، فيمكن استخدام الواجهة FileReader من أجل الوصول إلى محتويات ذلك الملف من برنامج جافاسكربت، كما يُستخدم الكائنان localStorage وsessionStorage لحفظ المعلومات حتى مع إعادة التحميل، إذ يحتفظ الكائن الأول بالبيانات احتفاظًا دائمًا أو إلى أن يقرر المستخدِم محوها؛ أما الثاني فيحفظها إلى حين إغلاق المتصفح.

18.17 تدريبات

18.17.1 التفاوض على المحتوى

أحد الأمور التي يفعلها بروتوكول HTTP هو التفاوض على المحتوى content negotiation، حيث تُستخدَم ترويسة الطلب Accept لإخبار الخادم بنوع المستند الذي يريده العميل، وتتجاهل العديد من الخوادم

هذه الترويسة، لكن إذا عرف الخادم عدة طرق لترميز أحد الموارد، فسينظر حينها في هذه الترويسة ويرسل نوع الملف الذي يريده العميل.

لقد هُيء الرابط https://eloquentJavaScript.net/author ليستجيب للنصوص المجردة أو HTML أو JSON وفقًا لما يطلبه العميل، وتعرَّف تلك الصيغ بأنواع وسائط قياسية هي `text/plain` و`text/html` و `application/json`.

أرسِل طلبات لجلب هذه الصيغ الثلاث من ذلك الرابط، واستخدم الخاصية `headers` في كائن الخيارات الممرَّر إلى `fetch` لضبط الترويسة `Accept` إلى نوع الوسائط media المفضل، ثم حاول طلب نوع الوسائط `application/rainbows+unicorns`، وانظر إلى شيفرة الحالة التي تنتجها.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
// شيفرتك هنا.
```

إرشادات الحل

- ابن شيفرتك على أمثلة `fetch` الموجودة في الفصل أعلاه.
- إذا طلبت أنواع وسائط كاذبة فستحصل على استجابة بالرمز 406، بمعنى "غير مقبول" أو Not acceptable، وهو الرمز الذي يجب أن يعيده الخادم إذا لم يستطع تحقيق الترويسة `Accept`.

18.17.2 طاولة عمل جافاسكربت

ابن واجهةً تسمح للناس بكتابة شيفرات جافاسكربت ويشغلونها، وضَع زرًا بجانب حقل `<textarea>`، بحيث إذا ضُغط عليه يمرِّر الباني `Function` الذي رأيناه في الفصل العاشر لتغليف النص في دالة واستدعائها، ثم حوِّل القيمة التي تعيدها الدالة أو أيّ خطأ ترفعه إلى سلسلة نصية واعرضها تحت الحقل النصي.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<textarea id="code">return "hi";</textarea>
<button id="button">Run</button>
<pre id="output"></pre>

<script>
  // شيفرتك هنا.
</script>
```

إرشادات الحل

استخدم `document.querySelector` أو `document.getElementById` من أجل الوصول إلى العناصر المعرَّفة في HTML لديك، وبالنسبة للحدَثين `"click"` و`"mousedown"`، فسيستطيع معالِج الحدث الحصول على الخاصية `value` للحقل النصي واستدعاء `Function` عليها، وتأكد من تغليف كل من الاستدعاء إلى `Function` والاستدعاء إلى نتيجته في كتلة `try` كي تستطيع التقاط الاستثناءات التي ترفعها، كما أننا لا نعرف في حالتنا هذه نوع الاستثناء الذي لدينا، لذا يجب التقاط كل شيء.

يمكن استخدام الخاصية `textContent` الخاصة بعنصر الخرج لملئها برسالة نصية؛ أما في حالة الرغبة في الحفاظ على المحتوى القديم، فأنشئ عقدةً نصيةً جديدةً باستخدام `document.createTextNode` وألحقها بالعنصر، وتذكّر إضافة محرف سطر جديد إلى النهاية كي لا يظهر كل الخرج على سطر واحد.

18.17.3 لعبة حياة كونويل

تُعَدّ لعبة حياة كونويل Conway game of life محاكاةً بسيطةً تنشئ حياةً صناعيةً على شبكة، بحيث تكون كل خلية في تلك الشبكة إما حيةً أو ميتةً، وتطبق القواعد التالية في كل جيل -منعطف-:

- تموت أيّ خلية حية لها أكثر من ثلاثة جيران أحياء أو أقل من اثنين.
- تبقى أيّ خلية حية على قيد الحياة حتى الجيل التالي إذا كان لها جاران أحياء أو ثلاثة.
- تعود أيّ خلية ميتة إلى الحياة إذا كان لها ثلاثة جيران أحياء حصرًا.

يُعرَّف الجار على أنه أيّ خلية مجاورة بما في ذلك الخلايا المجاورة له قطريًا. لاحظ أنّ تلك القواعد تطبَّق على كامل الشبكة مرةً واحدةً وليس على مربع واحد في المرة، وهذا يعني أنّ عدد الجيران مبني على الموقف الابتدائي، ولا تؤثر التغيرات الحادثة في الخلايا المجاورة في هذا الجيل على الحالة الجديدة للخلية.

نفِّذ هذه اللعبة باستخدام أيّ هيكل بيانات تجده مناسبًا، واستخدم `Math.random` لتوليد عشوائي لأماكن الخلايا في الشبكة في أول مرة، واعرضها على هيئة شبكة من حقول أزرار الاختيار مع زر بجانبها للانتقال إلى الجيل التالي، كما يجب عند حساب الجيل التالي إدراج التغييرات الحادثة عند اختيار المستخدم لزر الاختيار أو إلغاء الاختيار له.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<div id="grid"></div>
<button id="next">Next generation</button>

<script>
```

```
    // شيفرتك هنا.
  </script>
```

إرشادات الحل

- حاول النظر إلى حساب الجيل على أنه دالة محضة تأخذ شبكةً واحدةً وتنتج شبكةً جديدةً تمثِّل الدورة التالية.
- يمكن تمثيل المصفوفة matrix بالطريقة المبينة في الفصل السادس، حيث تَعد الجيران الأحياء بحلقتين تكراريتين متشعبتَين تكرران على إحداثيات متجاورة في كلا البعدين.
- لا تَعد الخلايا التي تكون خارج الحقل وتجاهل الخلية التي تكون في المركز عند عَد خلايا مجاورة لها.
- يمكن ضمان حدوث التغييرات على أزرار الاختيار في الجيل التالي بطريقتين؛ إما أن يلاحظ معالج حدث تلك التغييرات ويحدِّث الشبكة الحالية لتوافق ذلك، أو نولد شبكةً جديدةً من القيم التي في أزرار الاختيار قبل حساب الدورة التالية.
- إذا اخترت أسلوب معالج الحدث فربما يجب عليك إلحاق سمات تعرِّف الموضع الموافق لكل زر اختيار كي يسهل معرفة الخلية التي يجب تغييرها؛ أما لرسم شبكة من أزرار الاختيار، فيمكن استخدام العنصر `<table>` -انظر الفصل الرابع عشر- أو وضعها جميعًا في العنصر نفسه ووضع عناصر `<br>` -أي فواصل الأسطر- بين الصفوف.

مستقل
mostaql.com

ادخل سوق العمل ونفذ المشاريع باحترافية
عبر أكبر منصة عمل حر بالعالم العربي

ابدأ الآن كمستقل

19. مشروع محرر رسوم نقطية

> إنني أنظر إلى هذه الألوان التي أمامي، ثم أنظر إلى اللوحة البيضاء، ثم أحاول وضع الألوان عليها كما يضع الشاعر كلماته في قصيدته تمامًا.
>
> — يوان ميرو Joan Miro.

سنبني في هذا الفصل تطبيق ويب بناءً على ما درسناه في الفصول السابقة وسيكون هذا التطبيق للرسم بأسلوب البكسلات، حيث يمكنك أخذ رؤية مكبرة من الصورة المنفردة وتغيير أو تعديل كل بكسل فيها، كما يمكنك فتح الصورة بالبرنامج وتعديلها بالفأرة أو أيّ أداة تأشير أخرى ثم حفظها، حيث سيبدو البرنامج هكذا:

19.1 المكونات

تُظهر واجهة البرنامج عنصر <canvas> كبيرًا في الأعلى مع عدد من حقول الاستمارات form fields أسفله، ويرسم المستخدِم على الصورة عبر اختيار أداة من حقل <select> ثم ينقر عليه أو يدهن أو يسحب المؤشر في لوحة الرسم، كما هناك أدوات لرسم بكسلات منفردة أو مستطيلات ولملء مساحة ما باللون ولالتقاط لون ما من الصورة.

سنبني هيكل المحرِّر على أساس عدد من المكونات والكائنات التي تكون مسؤولة عن جزء من تمثيل كائن المستند DOM وقد تحتوي مكونات أخرى داخلها، كما تتكون حالة التطبيق من الصورة الحالية والأداة المختارة واللون المختار كذلك، حيث سنضبط هذه المتغيرات كي تكون الحالة داخل قيمة واحدة، كما تبني مكونات الواجهة مظهرها دائمًا على الحالة الراهنة.

دعنا ننظر في بديل ذلك كي نرى أهميته من خلال توزيع أجزاء من الحالة على الواجهة، وهذا سهل على البرنامج حتى نقطة ما، حيث نستطيع وضع حقل اللون ونقرأ قيمته عندما نريد معرفة اللون الحالي، لكن نضيف هنا منتقي الألوان color picker الذي يُعَدّ الأداة التي تسمح لك بالنقر على الصورة لاختيار اللون من بكسل ما، ولكي يظل حقل اللون مُظهرًا اللون الصحيح يجب على هذه الأداة معرفة أنه موجود وتحدِّثه كلما اختارت لونًا جديدًا، فإذا أضفت مكانًا آخرًا يجعل اللون مرئيًا بحيث يستطيع مؤشر الفأرة إظهاره مثلًا، فعليك تحديث شيفرة تغيير اللون لديك كي تبقى متزامنة.

لكن في الواقع يخلق هذا مشكلةً بحيث يحتاج كل جزء من الواجهة إلى معرفة جميع الأجزاء الأخرى، وهذا ليس عمليًا إذ سيجعل البرنامج أقل مرونة في التعديل عليه، لكن لا يمثِّل هذا مشكلةً بالنسبة لبرنامج صغير مثل برنامجنا؛ أما في المشاريع الكبيرة فسيكون هذا كارثة حقيقية، ولكي نتجنب هذا الكابوس وإن كان من حيث المبدأ في مثالنا فسنكون حازمين بشأن تدفق البيانات، فهناك حالة تُرسم الواجهة وفقًا لها، وقد يستجيب مكوِّن الواجهة لإجراءات المستخدِم عبر تحديث الحالة، بحيث تحصل المكونات عندئذ على فرصة لمزامنة أنفسها مع هذه الحالة الجديدة.

ضُبِط كل مكون من الناحية العملية ليُخطر عناصره الفرعية بإشعار كلما أُعطي حالةً جديدةً بالقدر الذي تحتاج إليه، لكن يُعَدّ ضبط ذلك أمرًا متعبًا، كما تفضِّل المتصفحات كثيرًا المكتبات البرمجية التي تسهل ذلك، لكن نستطيع إنشاء برنامج صغير مثل هذا بدون هذه البنية التحتية، كما تمثَّل التحديثات على الحالة على أساس كائنات سنطلق عليها إجراءات، وقد تنشئ المكونات مثل هذه الإجراءات وترسلها بسرعة إلى دالة مركزية لإدارة الحالة، حيث تحسب هذه الدالة الحالة التالية ثم تحدِّث مكونات الواجهة أنفسها إليها.

نأخذ بهذا مهمة تشغيل واجهة المستخدِم ونضيف إليها بعض الهيكلة، كما تحتفظ الأجزاء المتعلقة بـ DOM بما يشبه العمود الفقري رغم أنها ملأى بالآثار الجانبية، إذ يُعَدّ هذا العمود دورة تحديث الحالة، كما تحدِّد الحالة كيف سيبدو DOM، ولا توجد طريقة تستطيع أحداث DOM تغيير الحالة بها إلا عبر إرسال الإجراءات إلى

الحالة، كما توجد هناك صور عدة لهذا المنظور ولكل منها منافعه ومساوئه، لكن الفكرة الرئيسية لها واحدة وهي أنّ تغيرات الحالة يجب عليها المرور على قناة واحدة معرَّفة جيدًا بدلًا من كونها في كل مكان.

ستكون مكوناتنا أصنافًا مطابقةً للواجهة، ويُعطى بانيها حالةً قد تكون حالة البرنامج كله أو قيمةً أصغر إذا لم يحتج الوصول إلى كل شيء، حيث يستخدِم هذا في بناء خاصية `dom`، ويُعَدّ هذا عنصر DOM الذي سيمثِّل المكوِّن، كما ستأخذ أغلب المنشئات قيمًا أخرى أيضًا لا تتغير مع الوقت مثل الدالة التي تستطيع استخدامها لإرسال إجراء ما، ويملك كل مكوِّن تابعًا `syncState` يُستخدَم لمزامنته مع قيمة الحالة الجديدة، حيث يأخذ التابع وسيطًا واحدًا وهو الحالة التي تكون من نوع الوسيط الأول نفسه لبانيها.

19.2 الحالة

ستكون حالة التطبيق كائنًا له الخاصيات `picture` و`tool` و`color`، كما ستكون الصورة نفسها كائنًا يخزِّن العرض والارتفاع ومحتوى البكسل للصورة، في حين تُخزَّن البكسلات في مصفوفة ثنائية عبر طريقة صنف المصفوفة matrix نفسها من الفصل السادس صفًا صفًا من الأعلى حتى الأسفل.

```
class Picture {
  constructor(width, height, pixels) {
    this.width = width;
    this.height = height;
    this.pixels = pixels;
  }
  static empty(width, height, color) {
    let pixels = new Array(width * height).fill(color);
    return new Picture(width, height, pixels);
  }
  pixel(x, y) {
    return this.pixels[x + y * this.width];
  }
  draw(pixels) {
    let copy = this.pixels.slice();
    for (let {x, y, color} of pixels) {
      copy[x + y * this.width] = color;
    }
    return new Picture(this.width, this.height, copy);
  }
}
```

نريد التمكّن من معاملة الصورة على أنها قيمة غير قابلة للتغير immutable لأسباب سنعود إليها لاحقًا في هذا الفصل، لكن قد نحتاج أحيانًا إلى تحديث مجموعة بكسلات في الوقت نفسه أيضًا، ولكي نفعل ذلك فإن الصنف له تابع `draw` يتوقع مصفوفةً من البكسلات المحدَّثة، إذ تكون كائنات لها خاصيات `x` و`y` و`color`، كما ينشئ صورةً جديدةً مغيّرًا بها هذه البكسلات، ويستخدِم ذلك التابع `slice` دون وسائط لنسخ مصفوفة البسكلات كلها، بحيث تكون البداية الافتراضية لـ slice هي 0 والنهاية الافتراضية هي طول المصفوفة.

يستخدِم التابع `empty` جزأين من وظائف المصفوفة لم نرهما من قبل، فيمكن استدعاء باني `Array` بعدد لإنشاء مصفوفة فارغة بطول محدَّد، ثم يمكن استخدام التابع `fill` بعدها لملء هذه المصفوفة بقيمة معطاة، وتُستخدَم هذه الأجزاء لإنشاء مصفوفة تحمل كل البكسلات فيه اللون نفسه.

تُخزن الألوان على أساس سلاسل نصية تحتوي على رموز ألوان CSS العادية، وهي التي تبدأ بعلامة الشباك # متبوعة بستة أرقام ست-عشرية، بحيث يكون اثنان فيها للمكون الأحمر واثنان للأخضر واثنان للأزرق، وقد يكون هذا مبهمًا نوعًا ما، لكنها الصيغة التي تستخدمِها HTML في حقول ألوانها، حيث يمكن استخدامها في خاصية `fillStyle` لسياق لوحة رسم، وهي كافية لنا في هذا البرنامج، كما يُكتب اللون الأسود أصفارًا على الصورة `#000000`، ويبدو اللون الوردي الزاهي هكذا `#ff00ff` بحيث تكون مكونات اللونين الأحمر والأزرق لها القيمة العظمى عند 255، فتُكتب `ff` بالنظام الست-عشري الذي يستخدِم a حتى f لتمثيل الأعداد من 10 حتى 15.

سنسمح للواجهة بإرسال الإجراءات على أساس كائنات لها خاصيات تتجاوز خاصيات الحالة السابقة، ويرسل حقل اللون كائنًا حين يغيره المستخدِم مثل `{color: field.value}` تحسِب منه دالة التحديث حالةً جديدةً.

```
function updateState(state, action) {
  return Object.assign({}, state, action);
}
```

يُعَدّ استخدام `Object.assign` لإضافة خصائص `state` إلى كائن فارغ أولًا ثم تجاوز بعضها بخصائص من `action`، استخدامًا شائعًا في شيفرات جافاسكربت التي تستخدِم كائنات غير قابلة للتغير على صعوبته في التعامل معه، والأسهل من ذلك استخدام عاملًا ثلاثي النقاط لتضمين جميع الخصائص من كائن آخر في تعبير الكائن، وذلك لا زال بعد في مراحل اعتماده الأخيرة، وإذا تم فسوف تستطيع كتابة `{...state, ...action}` بدلًا من ذلك، لكن هذا لم يثبت عمله بعد في جميع المتصفحات.

19.3 بناء DOM

أحد الأمور التي تفعلها مكونات الواجهة هي إنشاء هيكل DOM، كما سنقدم نسخة موسعة قليلًا من دالة `elt` لأننا لا نريد استخدام توابع DOM في ذلك:

```
function elt(type, props, ...children) {
  let dom = document.createElement(type);
  if (props) Object.assign(dom, props);
  for (let child of children) {
    if (typeof child != "string") dom.appendChild(child);
    else dom.appendChild(document.createTextNode(child));
  }
  return dom;
}
```

الفرق الأساسي بين هذه النسخة والتي استخدمناها في الفصل السادس عشر أنها تسند خاصيات إلى عقد DOM وليس سمات، ويعني هذا أننا لا نستطيع استخدامها لضبط سمات عشوائية، لكن نستطيع استخدامها لضبط خاصيات قيمها ليست سلاسل نصية مثل `onclick` والتي يمكن تعيينها إلى دالة لتسجيل معالج حدث نقرة، وهذا يسمح بالنمط التالي من تسجيل معالِجات الأحداث:

```
<body>
  <script>
    document.body.appendChild(elt("button", {
      onclick: () => console.log("click")
    }, "The button"));
  </script>
</body>
```

19.4 اللوحة Canvas

يُعَدّ جزء الواجهة الذي يعرض الصورة على أساس شبكة من الصناديق المربعة المكوّن الأول الذي سنعرِّفه، وهذا الجزء مسؤول عن أمرين فقط هما عرض showing الصورة وتوصيل أحداث المؤشر التي على هذه الصورة إلى بقية التطبيق، وبالتالي يمكننا تعريفه على أساس مكوِّن لا يطلع إلا على الصورة الحالية وليس له شأن بحالة التطبيق كاملًا، كما لا يستطيع إرسال إجراءات مباشرةً لأنه لا يعرف كيف يعمل التطبيق، وأنما يستدعي دالة رد نداء callback function توفرها الشيفرة التي أنشأته حين يستجيب لأحداث المؤشر بحيث تعالِج أجزاء التطبيق على حدة.

```
const scale = 10;

class PictureCanvas {
  constructor(picture, pointerDown) {
```

```
    this.dom = elt("canvas", {
      onmousedown: event => this.mouse(event, pointerDown),
      ontouchstart: event => this.touch(event, pointerDown)
    });
    this.syncState(picture);
  }
  syncState(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  }
}
```

سنرسم كل بكسل على أساس مربع بعداه 10×10 كما هو مُحدَّد في ثابت `scale`، ثم يحتفظ المكوِّن بصورته الحالية ولا يعيد الرسم إلا حين تُعطى `syncState` صورةً جديدةً، كما تضبط دالة الرسم الفعلية حجم اللوحة وفقًا لمقياس الصورة وحجمها، ثم تملؤها بسلسلة من المربعات يمثِّل كل منها بكسلًا واحدًا.

```
function drawPicture(picture, canvas, scale) {
  canvas.width = picture.width * scale;
  canvas.height = picture.height * scale;
  let cx = canvas.getContext("2d");

  for (let y = 0; y < picture.height; y++) {
    for (let x = 0; x < picture.width; x++) {
      cx.fillStyle = picture.pixel(x, y);
      cx.fillRect(x * scale, y * scale, scale, scale);
    }
  }
}
```

إذا ضغطنا زر الفأرة الأيسر أثناء وجود الفأرة فوق لوحة الصورة فيستدعي المكون رد نداء `pointerDown` ليعطيه موضع البكسل الذي تم النقر عليه في إحداثيات الصورة، حيث سيُستخدم هذا لتنفيذ تفاعل الفأرة مع الصورة، وقد يعيد رد النداء دالة رد نداء أخرى لتُنبَّه بإشعار حين يتحرك المؤشر إلى بكسل آخر أثناء الضغط على الزر.

```
PictureCanvas.prototype.mouse = function(downEvent, onDown) {
  if (downEvent.button != 0) return;
```

```
    let pos = pointerPosition(downEvent, this.dom);
    let onMove = onDown(pos);
    if (!onMove) return;
    let move = moveEvent => {
      if (moveEvent.buttons == 0) {
        this.dom.removeEventListener("mousemove", move);
      } else {
        let newPos = pointerPosition(moveEvent, this.dom);
        if (newPos.x == pos.x && newPos.y == pos.y) return;
        pos = newPos;
        onMove(newPos);
      }
    };
    this.dom.addEventListener("mousemove", move);
  };

  function pointerPosition(pos, domNode) {
    let rect = domNode.getBoundingClientRect();
    return {x: Math.floor((pos.clientX - rect.left) / scale),
            y: Math.floor((pos.clientY - rect.top) / scale)};
  }
```

بما أننا نعرف حجم البكسلات ونستطيع استخدام `getBoundingClientRect` في إيجاد موضع اللوحة على الشاشة، فمن الممكن الذهاب من إحداثيات حدث الفأرة `clientX` و`clientY` إلى إحداثيات الصورة، وتُقرَّب هذه دومًا كي تشير إلى بكسل بعينه؛ أما بالنسبة لأحداث اللمس فيتوجب علينا فعل شيء قريب من ذلك لكن باستخدام أحداث مختلفة والتأكد أننا نستدعي `preventDefault` على حدث `"touchstart"` لمنع التمرير العمودي أو الأفقي panning.

```
  PictureCanvas.prototype.touch = function(startEvent,
                                           onDown) {
    let pos = pointerPosition(startEvent.touches[0], this.dom);
    let onMove = onDown(pos);
    startEvent.preventDefault();
    if (!onMove) return;
    let move = moveEvent => {
      let newPos = pointerPosition(moveEvent.touches[0],
```

```
                                    this.dom);
      if (newPos.x == pos.x && newPos.y == pos.y) return;
      pos = newPos;
      onMove(newPos);
    };
    let end = () => {
      this.dom.removeEventListener("touchmove", move);
      this.dom.removeEventListener("touchend", end);
    };
    this.dom.addEventListener("touchmove", move);
    this.dom.addEventListener("touchend", end);
  };
```

لا تكون أحداث `clientX` و`clientY` متاحةً مباشرةً لأحداث اللمس على كائن الحدث، لكن نستطيع استخدام إحداثيات كائن اللمس الأول في خاصية `touches`.

19.5 التطبيق

سننفذ المكون الأساسي على أساس صدَفة حول لوحة الصورة كي نبني التطبيق جزءًا جزءًا مع مجموعة من الأدوات والمتحكمات التي نمررها لبانيه، كما ستكون المتحكمات عناصر الواجهة التي ستظهر تحت الصورة، وستكون متاحةً في صورة مصفوفة من بواني المكونات.

تفعل الأدوات مهامًا مثل رسم البكسلات أو ملء مساحة ما، ويعرض التطبيق مجموعةً من الأدوات المتاحة مثل حقل `<select>`، كما تحدِّد الأداة المختارة حاليًا ما يحدث عندما يتفاعل المستخدِم مع الصورة بأداة تأشير مثل الفأرة، وتوفِّر مجموعة من الأدوات المتاحة على أساس كائن ينظِّم الأسماء التي تظهر في الحقل المنسدل للدوال التي تنفِّذ الأدوات، كما تحصل مثل هذه الدوال على موضع الصورة وحالة التطبيق الحالية ودالة `dispatch` في هيئة وسائط، وقد تعيد دالة معالِج حركة move handler تُستدعى مع موضع جديد وحالة حالية عندما يتحرك المؤشر إلى بكسل جديد.

```
class PixelEditor {
  constructor(state, config) {
    let {tools, controls, dispatch} = config;
    this.state = state;

    this.canvas = new PictureCanvas(state.picture, pos => {
      let tool = tools[this.state.tool];
```

```
      let onMove = tool(pos, this.state, dispatch);
      if (onMove) return pos => onMove(pos, this.state);
    });
    this.controls = controls.map(
      Control => new Control(state, config));
    this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                   ...this.controls.reduce(
                     (a, c) => a.concat(" ", c.dom), []));
  }
  syncState(state) {
    this.state = state;
    this.canvas.syncState(state.picture);
    for (let ctrl of this.controls) ctrl.syncState(state);
  }
}
```

يستدعي معالج المؤشر المعطى لـ `PictureCanvas` الأداة المختارة حاليًا باستخدام الوسائط المناسبة، وإذا أعاد معالج حركة فسيكيّفه ليستقبل الحالة، وتُنشأ جميع المتحكمات وتُخزَّن في `this.controls` كي يمكن تحديثها حين تتغير حالة التطبيق، ويدخل استدعاء `reduce` مسافات بين عناصر متحكمات DOM كي لا تبدو هذه العناصر مكثَّفة بجانب بعضها، كما تُعَدّ قائمة اختيار الأدوات أول متحكم، وتنشئ عنصر `<select>` مع خيار لكل أداة وتضبط معالِج حدث `"change"` الذي يحدِّث حالة التطبيق حين يختار المستخدِم أداةً مختلفةً.

```
class ToolSelect {
  constructor(state, {tools, dispatch}) {
    this.select = elt("select", {
      onchange: () => dispatch({tool: this.select.value})
    }, ...Object.keys(tools).map(name => elt("option", {
      selected: name == state.tool
    }, name)));
    this.dom = elt("label", null, "🖌 Tool: ", this.select);
  }
  syncState(state) { this.select.value = state.tool; }
}
```

حين نغلِّف نص العنوان text والحقل داخل عنصر `<label>` فإننا نخبر المتصفح أن العنوان ينتمي إلى هذا الحقل كي تستطيع النقر مثلًا على العنوان لتنشيط الحقل، كذلك نحتاج إلى إمكانية تغيير اللون، لذا

سنضيف متحكمًا لهذا وهو عنصر <input> من HTML مع سمة type لـ color، بحيث تعطينا حقل استمارة مخصص لاختيار الألوان، وقيمةً مثل هذا الحقل تكون دائمًا رمز لون CSS بصيغة "#RRGGBB" أي الأحمر ثم الأخضر ثم الأزرق بمعنى رقمين لكل لون، وسيعرض المتصفح واجهة مختار الألوان color picker عندما يتفاعل المستخدِم معها، كما ينشئ هذا المتحكم مثل ذلك الحقل ويربطه ليكون متزامنًا مع خاصية color الخاصة بحالة التطبيق.

```
class ColorSelect {
  constructor(state, {dispatch}) {
    this.input = elt("input", {
      type: "color",
      value: state.color,
      onchange: () => dispatch({color: this.input.value})
    });
    this.dom = elt("label", null, "🎨 Color: ", this.input);
  }
  syncState(state) { this.input.value = state.color; }
}
```

19.6 أدوات الرسم

نحتاج قبل رسم أي شيء إلى تنفيذ الأدوات التي ستتحكم في وظائف الفأرة وأحداث اللمس على اللوحة، وأبسط أداة هي أداة الرسم التي تغير أيّ بكسل تنقر عليه أو تلمسه بإصبعك إلى اللون الحالي، وترسل إجراءً يحدِّث الصورة إلى إصدار يُعطى فيه البكسل المشار إليه اللون المختار الحالي.

```
function draw(pos, state, dispatch) {
  function drawPixel({x, y}, state) {
    let drawn = {x, y, color: state.color};
    dispatch({picture: state.picture.draw([drawn])});
  }
  drawPixel(pos, state);
  return drawPixel;
}
```

تستدعي الدالة فورًا دالة drawPixel ثم تعيدها كي تُستدعى مرةً أخرى من أجل البكسلات التي ستُلمَس لاحقًا حين يسحب المستخدِم إصبعه أو يمرره على الصورة، ولكي نرسم أشكالًا أكبر فمن المفيد إنشاء مستطيلات بسرعة، كما ترسم أداة rectangle مستطيلًا بين النقطة التي تبدأ السحب منها حتى النقطة التي تترك فيها المؤشر أو ترفع إصبعك.

```
function rectangle(start, state, dispatch) {
  function drawRectangle(pos) {
    let xStart = Math.min(start.x, pos.x);
    let yStart = Math.min(start.y, pos.y);
    let xEnd = Math.max(start.x, pos.x);
    let yEnd = Math.max(start.y, pos.y);
    let drawn = [];
    for (let y = yStart; y <= yEnd; y++) {
      for (let x = xStart; x <= xEnd; x++) {
        drawn.push({x, y, color: state.color});
      }
    }
    dispatch({picture: state.picture.draw(drawn)});
  }
  drawRectangle(start);
  return drawRectangle;
}
```

هناك تفصيل مهم في هذا التنفيذ، وهو أنك حين تسحب المؤشر سيعاد رسم المستطيل على الصورة من الحالة الأصلية، وهكذا تستطيع جعل المستطيل أكبر أو أصغر مرةً أخرى أثناء إنشائه دون مستطيل وسيط يتبقى في الصورة النهائية، وهذا أحد الأسباب التي تجعل كائنات الصورة غير القابلة للتغيّر مفيدةً، كما سننظر في سبب آخر لاحقًا، وسيكون تنفيذ مهمة ملء اللون أكثر تفصيلًا، إذ هي أداة تملأ البكسل الذي تحت المؤشر والبكسلات المجاورة له التي لها اللون نفسه، وإنما نعني بالمجاورة له تلك البكسلات المجاورة رأسيًا أو عموديًا مباشرةً وليس البكسلات المجاورة قطريًا له، كما توضِّح الصورة التالية مجموعة بكسلات تُلوَّن باستخدام أداة الملء على البكسل الذي يحمل النقطة الحمراء.

من المثير أن الطريقة التي نفعل بها ذلك تشبه شيفرة الاستكشاف التي تعرضنا لها في الفصل السابع، حيث بحثت تلك الشيفرة في مخطط لإيجاد طريق ما للروبوت، وتبحث هذه الشيفرة في شبكة لإيجاد كل البكسلات المرتبطة ببعضها بعضًا، لكن مشكلة تتبع مجموعة الفروع الممكنة مشابهةً هنا.

```
const around = [{dx: -1, dy: 0}, {dx: 1, dy: 0},
                {dx: 0, dy: -1}, {dx: 0, dy: 1}];

function fill({x, y}, state, dispatch) {
  let targetColor = state.picture.pixel(x, y);
  let drawn = [{x, y, color: state.color}];
  for (let done = 0; done < drawn.length; done++) {
    for (let {dx, dy} of around) {
      let x = drawn[done].x + dx, y = drawn[done].y + dy;
      if (x >= 0 && x < state.picture.width &&
          y >= 0 && y < state.picture.height &&
          state.picture.pixel(x, y) == targetColor &&
          !drawn.some(p => p.x == x && p.y == y)) {
        drawn.push({x, y, color: state.color});
      }
    }
  }
  dispatch({picture: state.picture.draw(drawn)});
}
```

تتصرف مصفوفة البكسلات المرسومة على أساس قائمة العمل للدالة، فيجب علينا من أجل كل بكسل نصل إليه رؤية إذا كان أيّ بكسل مجاور له يحمل اللون نفسه ولم يُدهن مسبقًا، وتتأخر حلقة العد التكرارية خلف طول مصفوفة `drawn` بسبب إضافة البكسلات الجديدة، كما سيحتاج أيّ بكسل يسبقها إلى استكشافه، وحين تلحق بالطول فستكون كل البكسلات قد استُكشِفت وقد أتمت الدالة عملها؛ أما الأداة النهائية فهي مختار الألوان color picker الذي يسمح لك بالإشارة إلى لون في الصورة لاستخدامه على أساس لون الرسم الحالي.

```
function pick(pos, state, dispatch) {
  dispatch({color: state.picture.pixel(pos.x, pos.y)});
}
```

نستطيع الآن اختبار التطبيق!

```
<div></div>
<script>
  let state = {
    tool: "draw",
    color: "#000000",
```

```
    picture: Picture.empty(60, 30, "#f0f0f0")
  };
  let app = new PixelEditor(state, {
    tools: {draw, fill, rectangle, pick},
    controls: [ToolSelect, ColorSelect],
    dispatch(action) {
      state = updateState(state, action);
      app.syncState(state);
    }
  });
  document.querySelector("div").appendChild(app.dom);
</script>
```

19.7 الحفظ والتحميل

لا شك أننا حين نرسم الصورة الخاصة بنا سنود حفظها لاحقًا، كما يجب إضافة زر لتحميل الصورة الحالية على أساس ملف صورة، حيث يوفر المتحكم التالي هذا الزر:

```
class SaveButton {
  constructor(state) {
    this.picture = state.picture;
    this.dom = elt("button", {
      onclick: () => this.save()
    }, "💾 Save");
  }
  save() {
    let canvas = elt("canvas");
    drawPicture(this.picture, canvas, 1);
    let link = elt("a", {
      href: canvas.toDataURL(),
      download: "pixelart.png"
    });
    document.body.appendChild(link);
    link.click();
    link.remove();
  }
  syncState(state) { this.picture = state.picture; }
```

```
}
```

يتتبّع المكون الصورة الحالية ليستطيع الوصول إليها عند الحفظ، كما يستخدِم عنصر `<canvas>` لإنشاء ملف الصورة والذي يرسم الصورة على مقياس بكسل واحد لكل بكسل، في حين ينشئ التابع `toDataURL` الذي على عنصر اللوحة رابطًا يبدأ بـ `data:` على عكس الروابط التي تبدأ ببروتوكولات `http:` و`https:` العادية، تحتوي هذه الروابط على المصدر كاملًا في الرابط، كما تكون طويلةً جدًا لهذا، لكنه يسمح لنا بإنشاء روابط عاملة إلى صور عشوائية من داخل المتصفح.

ننشئ عنصر رابط للوصول إلى المتصفح وتحميل الصورة يشير إلى هذا الرابط وله سمة `download`، وعندما يُنقَر على مثل هذه الروابط فستجعل المتصفح يعرض صندوقًا حواريًا لحفظ الملف، كما نضيف ذلك الرابط إلى المستند ونحاكي النقر عليه ثم نحذفه مرةً أخرى، وهكذا تستطيع فعل الكثير والكثير باستخدام التقنيات المتاحة في المتصفح لكن قد تبدو بعض هذه التقنيات غريبةً، بل إذا أردنا أن نكون قادرين على تحميل ملفات صورة موجودة إلى تطبيقنا، فسنحتاج إلى تعريف مكوِّن لزر، أي كما في المثال التالي:

```
class LoadButton {
  constructor(_, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => startLoad(dispatch)
    }, "🗁 Load");
  }
  syncState() {}
}

function startLoad(dispatch) {
  let input = elt("input", {
    type: "file",
    onchange: () => finishLoad(input.files[0], dispatch)
  });
  document.body.appendChild(input);
  input.click();
  input.remove();
}
```

إذا أردنا الوصول إلى ملف في حاسوب المستخدِم، فسيكون على المستخدِم اختيار الملف من حقل إدخال الملف، لكنّا لا نريد أن يبدو زر التحميل مثل حقل إدخال ملف، لذا سننشئ إدخال الملف عندما يُنقر على الزر ونتظاهر حينها أنّ إدخال الملف ذاك قد نُقر عليه، فإذا اختار المستخدِم ملفًا، فسنستطيع استخدام

FileReader للوصول إلى محتوياته في صورة رابط بيانات كما ذكرنا قبل قليل، ويمكن استخدام هذا الرابط لإنشاء عنصر <img>، لكن بسبب أننا لا نستطيع الوصول مباشرةً إلى البكسلات في مثل هذه الصورة فلا نستطيع إنشاء كائن Picture منها.

```
function finishLoad(file, dispatch) {
  if (file == null) return;
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    let image = elt("img", {
      onload: () => dispatch({
        picture: pictureFromImage(image)
      }),
      src: reader.result
    });
  });
  reader.readAsDataURL(file);
}
```

يجب علينا رسم الصورة أولًا في عنصر <canvas> كي نصل إلى البكسلات، كما يملك سياق اللوحة canvas التابع getImageData الذي يسمح للسكربت قراءة بكسلاتها، لذا بمجرد أن تكون الصورة على اللوحة يمكننا الوصول إليها وبناء كائن Picture.

```
function pictureFromImage(image) {
  let width = Math.min(100, image.width);
  let height = Math.min(100, image.height);
  let canvas = elt("canvas", {width, height});
  let cx = canvas.getContext("2d");
  cx.drawImage(image, 0, 0);
  let pixels = [];
  let {data} = cx.getImageData(0, 0, width, height);

  function hex(n) {
    return n.toString(16).padStart(2, "0");
  }
  for (let i = 0; i < data.length; i += 4) {
    let [r, g, b] = data.slice(i, i + 3);
    pixels.push("#" + hex(r) + hex(g) + hex(b));
```

```
    }
    return new Picture(width, height, pixels);
  }
```

سنحدّ من حجم الصور إلى أن تكون 100×100 بكسل، بما أن أي شيء أكبر من هذا سيكون أكبر من أن يُعرض على الشاشة وقد يبطئ الواجهة، كما تكون خاصية `data` الخاصة بالكائن الذي يعيده `getImageData` مصفوفةً من مكونات الألوان، إذ تحتوي على أربع قيم لكل بكسل في المستطيل الذي تحدده الوسائط، حيث تمثل مكونات البكسل اللونية من الأحمر والأخضر والأزرق والشفافية alpha، كما تكون هذه المكونات أرقامًا تتراوح بين الصفر و255، ويعني الصفر في خانة الألفا أنه شفاف تمامًا و255 أنه مصمت، لكن سنتجاهل هذا في مثالنا إذ لا يهمنا كثيرًا.

يتوافق كل رقمين ست-عشريين لكل مكوِّن مستخدَم في ترميزنا للألوان توافقًا دقيقًا للمجال الذي يتراوح بين الصفر و255، حيث يستطيع هذان الرقمان التعبير عن 256 = 162 عددًا، كما يمكن إعطاء القاعدة إلى التابع `toString` الخاص بالأعداد على أساس وسيط كي ينتج `n.toString(16)` تمثيلًا من سلسلة نصية في النظام الست عشري، ويجب التأكد من أنّ كل عدد يأخذ رقمين فقط، لذلك فإن الدالة المساعِدة `hex` تستدعي `padStart` لإضافة صفر بادئ عند الحاجة، ونستطيع الآن التحميل والحفظ ولم يبق إلا ميزة إضافية واحدة.

19.8 سجل التغييرات Undo History

ستكون نصف عملية التعديل على الصور بارتكاب أخطاء صغيرة بين الحين والآخر ثم تصحيحها، لذا من المهم لنا وجود سجل للخطوات التي ينفذها المستخدِم كي يستطيع العودة إليها وتصحيح ما يريده، حيث سنحتاج لهذا تخزين النسخ السابقة من الصورة، وهو أمر يسير بما أنها قيمة غير قابلة للتغيّر لكنها تحتاج حقلًا إضافيًا داخل حالة التطبيق.

سنضيف مصفوفة `done` للحفاظ على النسخ السابقة من الصورة، كما سيتطلب الحفاظ على هذه الخاصية دالةً معقدةً لتحديث الحالة بحيث تضيف الصورة إلى المصفوفة، لكن لا نريد تخزين كل تغيير يحدث، وإنما التغييرات التي تحدث كل فترة زمنية محدَّدة، لذا سنحتاج إلى خاصية ثانية هي `doneAt` تتتبّع آخر وقت حفظنا فيه صورة في السجل.

```
function historyUpdateState(state, action) {
  if (action.undo == true) {
    if (state.done.length == 0) return state;
    return Object.assign({}, state, {
      picture: state.done[0],
```

```
        done: state.done.slice(1),
        doneAt: 0
      });
    } else if (action.picture &&
               state.doneAt < Date.now() - 1000) {
      return Object.assign({}, state, action, {
        done: [state.picture, ...state.done],
        doneAt: Date.now()
      });
    } else {
      return Object.assign({}, state, action);
    }
  }
```

إذا كان الإجراء هو إجراء تراجع undo، فستأخذ الدالة آخر صورة من السجل وتجعلها هي الصورة الحالية، حيث تضبط doneAt على صفر كي نضمن أن التغيير التالي يخزِّن الصورة في السجل مما يسمح لك بالعودة إليها في وقت آخر إذا أردت؛ أما إن كان الإجراء غير ذلك ويحتوي على صورة جديدة وكان آخر وقت تخزين صورة أكثر من ثانية واحدة -أي أكثر من 1000 ميلي ثانية-، فستُحدَّث خصائص done وdoneAt لتخزين الصورة السابقة، كما لا يفعل مكوّن زر التراجع الكثير، إذ يرسِل إجراءات التراجع عند النقر عليه ويعطِّل نفسه إذا لم يكن ثمة شيء يُتراجع عنه.

```
class UndoButton {
  constructor(state, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => dispatch({undo: true}),
      disabled: state.done.length == 0
    }, "⮪ Undo");
  }
  syncState(state) {
    this.dom.disabled = state.done.length == 0;
  }
}
```

19.9 لنرسم

نحتاج أولًا إلى إنشاء حالة كي نستطيع استخدام التطبيق مع مجموعة من الأدوات والمتحكمات ودالة إرسال، ثم نمرر إليها باني `PixelEditor` لإنشاء المكوِّن الأساسي، وبما أننا نحتاج إلى إنشاء عدة محررات في التدريبات، فسنعرِّف بعض الرابطات bindings أولًا.

```
const startState = {
  tool: "draw",
  color: "#000000",
  picture: Picture.empty(60, 30, "#f0f0f0"),
  done: [],
  doneAt: 0
};

const baseTools = {draw, fill, rectangle, pick};

const baseControls = [
  ToolSelect, ColorSelect, SaveButton, LoadButton, UndoButton
];

function startPixelEditor({state = startState,
                           tools = baseTools,
                           controls = baseControls}) {
  let app = new PixelEditor(state, {
    tools,
    controls,
    dispatch(action) {
      state = historyUpdateState(state, action);
      app.syncState(state);
    }
  });
  return app.dom;
}
```

تستطيع استخدام `=` بعد اسم الرابطة حين نفكك كائنًا أو مصفوفةً كي تعطي الرابطة قيمةً افتراضيةً تُستخدَم عندما تكون الخاصية مفقودةً أو تحمل قيمة غير معرفة `undefined`، كما تستفيد دالة `StartPixelEditor` من هذا في قبول كائن له عدد من الخصائص الاختيارية على أساس وسيط، فإذا لم

توفر خاصية `tools`، فستكون مقيدةً إلى `baseTools`، وتوضِّح الشيفرة التالية كيفية الحصول على محرر حقيقي على الشاشة:

```
<div></div>
<script>
  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

تستطيع الآن الرسم فيه إذا شئت.

19.10 سبب صعوبة البرنامج

لا شك أنّ التقنيات المتعلقة بالمتصفحات رائعة وتمكننا من فعل الكثير بالواجهات المرئية من بنائها وتعديلها بل وتنقيحها من الأخطاء البرمجية كذلك، كما ستضمن أنّ البرنامج الذي تكتبه من أجل المتصفح سيعمل على كل حاسوب وهاتف يتصل بالإنترنت، لكن تُعَدّ تقنيات المتصفحات هذه بحارًا واسعةً، إذ عليك تعلّم الكثير الكثير من الطرق والأدوات لتستطيع الاستفادة منها، كما أنّ النماذج البرمجية الافتراضية المتاحة لها كثيرة المشاكل إلى حد أن أغلب المبرمجين يفضلون التعامل مع طبقات مجرَّدة عليها عوضًا عن التعامل المباشر معها، وعلى الرغم من اتجاه الوضع نحو الأفضل، إلا أن هذا يكون في صورة إضافة مزيد من العناصر لحل المشاكل وأوجه القصور الموجودة مما يخلق المزيد من التعقيد.

لا يمكن استبدال الميزة المستخدَمة من قِبل ملايين المواقع بقرار واحد بسهولة، بل حتى لو أمكن ذلك فمن الصعب الاتفاق على بديلها، ونحن مقيدون بأدواتنا والعوامل الاجتماعية والاقتصادية والتاريخية التي أثرت في إنشائها، والأمر الذي قد يفيدنا في هذا هو أننا قد نصبح أفضل في الإنتاجية إذا عرفنا كيف تعمل هذه التقنيات ولماذا هي على الوجه التي عليه بدلًا من الثورة عليها وتجنبها كليًا.

قد تكون التجريدات الجديدة مفيدةً حقًا، فقد كان نموذج المكونات وأسلوب تدفق البيانات اللذان استخدمناهما في هذا الفصل مثالًا على ذلك، وهناك الكثير من المكتبات التي تجعل برمجة واجهة المستخدِم أفضل وأسهل، سيما React و Angular الشائعتا الاستخدام وقت كتابة هذا الكتاب، لكن هذا مجال كبير بحد ذاته ننصحك بإنفاق بعض الوقت في تصفّحه كي تعرف كيف تعمل هذه المكتبات والفوائد التي تجنيها منها.

19.11 تدريبات

لا زال هناك مساحة نطور فيها برنامجنا، فلمَ لا نضيف بعض المزايا الجديدة في صورة تدريبات؟

19.11.1 رابطات لوحة المفاتيح

أضف اختصارات للوحة المفاتيح إلى التطبيق، بحيث إذا ضغطنا على الحرف الأول من اسم أداة فستُختار الأداة، وctrl+z يفعِّل إجراء التراجع.

افعل ذلك عبر تعديل مكوِّن `PixelEditor` وأضف خاصية `tabIndex` التي تساوي 0 إلى العنصر المغلِّف `<div>` كي يستطيع استقبال التركيز من لوحة المفاتيح.

لاحظ أن الخاصية الموافقة لسمة `tabindex` تُسمى `tabIndex` حيث يكون حرف `I` فيها على الصورة الكبيرة، كما تتوقع دالة `elt` أسماء خصائص، ثم سجِّل معالجات أحداث المفاتيح مباشرةً على ذلك العنصر، حيث سيعني هذا أنه عليك ضغط أو لمس أو نقر التطبيق قبل أن تستطيع التفاعل معه بلوحة المفاتيح.

تذكَّر أنّ أحداث لوحة المفاتيح لها الخاصيتان `ctrlKey` و`metaKey` -المخصص لزر command في ماك-، حيث تستطيع استخدامهما لتعرف هل هذان الزران مضغوط عليهما أم لا.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<div></div>
<script>
  // الأصلي PixelEditor صنف.
  // وسع المنشئ.
  class PixelEditor {
    constructor(state, config) {
      let {tools, controls, dispatch} = config;
      this.state = state;

      this.canvas = new PictureCanvas(state.picture, pos => {
        let tool = tools[this.state.tool];
        let onMove = tool(pos, this.state, dispatch);
        if (onMove) {
          return pos => onMove(pos, this.state, dispatch);
        }
      });
```

```
        this.controls = controls.map(
          Control => new Control(state, config));
        this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                       ...this.controls.reduce(
                         (a, c) => a.concat(" ", c.dom), []));
      }
      syncState(state) {
        this.state = state;
        this.canvas.syncState(state.picture);
        for (let ctrl of this.controls) ctrl.syncState(state);
      }
    }

    document.querySelector("div")
      .appendChild(startPixelEditor({}));
  </script>
```

إرشادات الحل

- ستكون خاصية `key` لأحداث مفاتيح الأحرف هي الحرف نفسه في حالته الصغرى إذا لم يكن زر shift مضغوطًا، لكن لا تهمنا أحداث المفاتيح التي فيها زر shift.
- يستطيع معالج `"keydown"` فحص كائن الحدث الخاص به ليرى إذا كان يطابق اختصارًا من الاختصارات، كما تستطيع الحصول على قائمة من الأحرف الأولى من كائن `tools` كي لا تضطر إلى كتابتها.
- إذا طابق حدث مفتاح اختصارًا ما، استدع `preventDefault` عليه وأرسل الإجراء المناسب.

19.11.2 الرسم بكفاءة

سيكون أغلب العمل الذي يفعله التطبيق أثناء الرسم داخل `drawPicture`، ورغم أنّ إنشاء حالة جديدة وتحديث بقية DOM لن يكلفنا كثيرًا، إلا أنّ إعادة رسم جميع البكسلات في اللوحة يمثِّل مهمةً ثقيلةً، لذا جِد طريقةً لتسريع تابع `syncState` الخاص بـ `PictureCanvas` عبر إعادة الرسم في حالة تغير البكسلات فقط.

تذكَّر أنّ `drawPicture` تُستخدَم أيضًا بواسطة زر الحفظ، فتأكد إذا غيرتها من أنك لا تخرِّب الوظيفة القديمة، أو أنشئ نسخةً جديدةً باسم مختلف، ولاحظ أنّ تغيير حجم عنصر `<canvas>` عبر ضبط خاصيتَي `width` و`height` له يتسبب في مسحه ليصير شفافًا مرةً أخرى.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<div></div>
<script>
  // غيّر هذا التابع
  PictureCanvas.prototype.syncState = function(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  };

  // ربما تود تغيير هذا أو استخدامه كذلك.
  function drawPicture(picture, canvas, scale) {
    canvas.width = picture.width * scale;
    canvas.height = picture.height * scale;
    let cx = canvas.getContext("2d");

    for (let y = 0; y < picture.height; y++) {
      for (let x = 0; x < picture.width; x++) {
        cx.fillStyle = picture.pixel(x, y);
        cx.fillRect(x * scale, y * scale, scale, scale);
      }
    }
  }

  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

إرشادات الحل

يمثّل هذا التدريب مثالًا ممتازًا لرؤية كيف تسرّع هياكل البيانات غير القابلة للتغير من الشيفرة، كما نستطيع الموازنة بين الصورة القديمة والجديدة بما أن لدينا كليهما وإعادة الرسم في حالة تغير لون البكسلات فقط، مما يوفر 99% من مهمة الرسم في الغالب.

اكتب دالة `updatePicture` جديدةً أو اجعل دالـة `drawPicture` تأخـذ وسـيطًا إضـافيًا قـد يكـون غـير معرَّف أو قد يكون الصورة السابقة، وتتحقق الدالة لكل بكسل مما إذا كانت الصورة السابقة قـد مـرت على هـذا الموضع باللون نفسه أم لا، كما تتخطى البكسل الذي تكون تلك حالته.

يجب عليـك تجنب `width` و`height` حين يكـون للصـورتين الجديـدة والقديمـة نفس الحجم لأن اللوحـة تُمسح حين يتغير حجمها، فإذا اختلفتا -وتلك ستكون حالتنا إذا حُمِّلت صورة جديدة- فيمكنك ضبط الرابطة التي تحمل الصورة القديمة على null بعد تغيير حجم اللوحة، إذ يجب ألا تتخطى أيّ بكسل بعد تغيير حجم اللوحة.

19.11.3 الدوائر

عرِّف أداةً اسمها `circle` ترسم دائرةً مصمتةً حين تسحب بالمؤشر، حيث يكـون مركـز الـدائرة عنـد نقطـة بداية السحب، ويُحدَّد نصف قطره بالمسافة المسحوبة.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفيـة المتصـفح إن كنت تقـرأ من متصـفح، أو بنسخها إلى codepen.

```
<div></div>
<script>
  function circle(pos, state, dispatch) {
    // ضع شيفرتك هنا
  }

  let dom = startPixelEditor({
    tools: Object.assign({}, baseTools, {circle})
  });
  document.querySelector("div").appendChild(dom);
</script>
```

إرشادات الحل

تستطيع النظر في أداة `rectangle` لتستقي منها إرشادًا لهـذا التـدريب، حيث سـتحتاج إلى إبقـاء الرسـم على صـورة البـدء بـدلًا من الصـورة الحاليـة عنـدما يتحـرك المؤشـر، ولكي تعـرف أيّ البكسـلات يجب تلوينهـا، استخدام نظرية فيثاغورس بحساب المسافة بين الموضع الحالي للمؤشر والموضع الابتدائي من خلال أخذ الجذر التربيعي `Math.sqrt` لمجموع تربيع `Math.pow(x, 2)` لفرق في إحداثيات x وتربيع الفرق في إحداثيات y.

كرر بعد ذلك على تربيع البكسلات حول نقطة البداية التي تكون جوانبهـا ضـعف نصـف القطـر على الأقـل، ولوّن تلك التي تكون داخل نصف قطر الدائرة باستخدام معادلة فيثاغورس مرةً أخرى لتعـرف بُعـدها عن المركـز، وتأكد من أنك لا تلون البكسلات التي تكون خارج حدود الصورة.

19.11.4 الخطوط المستقيمة

يُعَدّ هذا التدريب متقدمًا أكثر مما قبله وسيحتاج إلى تصميم حل لمشكلة ليست بالهينة، لذا تأكد من امتلاكك وقت وصبر قبل أن تبدأ العمل عليه، ولا يمنعنك الفشل في المحاولات أن تعيد الكرة.

عندما تختار أداة `draw` في أغلب المتصفحات وتسحب المؤشر بسرعة ستجد أن ما حصلت عليه خطًا من النقاط التي تفصل بينها مسافات فارغة، وذلك لأن حدثَي `"mousemove"` أو `"touchmove"` لا ينطلقان بسرعة تغطي كل بكسل تمر عليه، لذا نريد منك تطوير أداة `draw` لتجعلها ترسم خطًا كاملًا، وهذا يعني أنه عليك جعل دالة معالج الحركة تتذكر الموضع السابق وتصله بالموضع الحالي، ولكي تفعل ذلك عليك كتابة دالة رسم خط عامة بما أنّ البكسلات التي تمر عليها قد لا تكون متجاورةً بما يصلح لخط مستقيم.

يُعَدّ الخط المستقيم بين بكسلين سلسلةً من البكسلات المتصلة في سلسلة واحدة بأقرب هيئة تمثل خطًا مستقيمًا من البداية إلى النهاية، ويُنظر إلى البكسلات المتجاورة قطريًا على أنها متصلة، لذا فإن الخط المائل يجب أن يبدو مثل الصورة التي على اليسار وليس الصورة اليمنى.

أخيرًا، إذا كانت لدينا شيفرةً ترسم خطًا بين نقطتين عشوائيتين فربما تريد استخدامها كي تعرِّف أداة سطر `line` ترسم خطًا مستقيمًا بين بداية السحب ونهايته.

تستطيع تعديل شيفرة التدريب لكتابة الحل وتشغيلها في طرفية المتصفح إن كنت تقرأ من متصفح، أو بنسخها إلى codepen.

```
<div></div>
<script>
  // هذه أداة الرسم القديمة، أعد كتابتها.
  function draw(pos, state, dispatch) {
    function drawPixel({x, y}, state) {
      let drawn = {x, y, color: state.color};
      dispatch({picture: state.picture.draw([drawn])});
    }
    drawPixel(pos, state);
```

```
    return drawPixel;
  }

  function line(pos, state, dispatch) {
    // ضع شيفرتك هنا.
  }

  let dom = startPixelEditor({
    tools: {draw, line, fill, rectangle, pick}
  });
  document.querySelector("div").appendChild(dom);
</script>
```

إرشادات الحل

تتكون مشكلة رسم خط من البكسلات من أربع مشكلات تختلف اختلافًا طفيفًا فيما بينها، إذ يُعَدّ رسم خط أفقي من اليسار إلى اليمين سهلًا إذ تكرر على إحداثيات x وتلون البكسل في كل خطوة، فإذا كان الخط يميل قليلًا أقل من 45 درجة أو ¼π راديان، فتستطيع وضع إحداثيات y مع الميل، لكن لا زلت في حاجة إلى بكسل لكل موضع x، ويُحدِّد الميل موضع y لكل بكسل من هذه البكسلات.

لكن ستحتاج إلى تغيير الطريقة التي تعامل بها الإحداثيات بمجرد تجاوز الميل درجة 45، حيث ستحتاج الآن إلى بكسل واحد لكل موضع y بما أن الخط يتقدم إلى الأعلى أكثر من سيره إلى اليسار، وعندما تتجاوز 135 درجة فعليك العودة إلى التكرار على إحداثيات x لكن من اليمين إلى اليسار.

لست بحاجة إلى كتابة أربع حلقات تكرارية، وبما أنّ رسم خط من A إلى B هو نفسه رسم خط من B إلى A، فيمكنك تبديل مواضع البداية والنهاية للخطوط التي تبدأ من اليمين إلى اليسار وتعامِلها على أنها من اليسار إلى اليمين، لذا تحتاج إلى حلقتين تكراريتين مختلفتين، وأول شيء يجب أن تفعله دالة رسم الخطوط هو معرفة هل الفرق بين إحداثيات x أكبر من الفرق بين إحداثيات y أم لا، فإذا كان فإنّ هذا خط مائل للأفقية، وإلا فإنه يميل لأن يكون رأسيًا.

تأكد من أن توازن بين القيم المطلقة لفروق x وy والتي تحصل عليها بواسطة `Math.abs`، وبمجرد معرفتك أيّ محور ستكرر عليه، تستطيع تفقد نقطة البدء لترى إذا كان لها إحداثي أعلى على هذا المحور من نقطة النهاية أم لا وتبدلهما إذا دعت الحاجة، وتكون الطريقة المختصرة هنا لتبديل قيم رابطتين في جافاسكربت تستخدِم مهمة تفكيك كما يلي:

```
[start, end] = [end, start];
```

ثم تحسب ميل الخط الذي يُحدِّد المقدار الذي يتغير به الإحداثي على المحور الآخر لكل خطوة تأخذها على المحور الأساسي، وتستطيع تشغيل حلقة تكرارية هنا على المحور الأساسي أثناء تتبع الموضع الموافق على المحور الآخر، كما يمكنك رسم بكسلات على كل تكرار، لكن تأكد من أن تقرِّب إحداثيات المحور غير الأساسي بما أنها ستكون أعدادًا كسرية على الأرجح ولا يتجاوب معها تابع `draw` جيدًا.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

20. بيئة Node.js: جافاسكربت خارج المتصفح

سأل أحد الطلاب: "استخدم المبرمجون قديمًا حواسيب بدائية ولم يستخدموا لغات برمجة، لكنهم أنتجوا برامج رائعة رغم هذا فلماذا كل هذا التعقيد الذي نراه اليوم في البرمجة ولغات البرمجة؟".

رد فو-تزو: استخدم البنّاءون الحطب والطين قديمًا، لكنهم بنوا رغم ذلك أكواخًا رائعةً.

— يوان-ما، كتاب البرمجة.

استخدمنا لغة جافاسكربت طيلة الفصول الماضية من الكتاب في بيئة واحدة هي بيئة المتصفح؛ أما في هذا الفصل والذي يليه فسنلقي نظرةً سريعةً على Node.js البرنامج الذي يسمح لك بتطبيق مهاراتك في جافاسكربت خارج نطاق المتصفح، إذ تستطيع بناء أيّ شيء بها من أدوات أوامر الطرفية البسيطة إلى خوادم HTTP التي تشغِّل مواقعًا ديناميكيةً، كما يهدف هذان الفصلان إلى شرح المفاهيم الأساسية التي تستخدمها Node.js وإعطاء معلومات تكفي لكتابة برامج لها، إلا أنهما لن يتعمقا في تفاصيل تلك المنصة.

لن تعمل أمثلة الشيفرات التي ستكون في هذا الفصل في المتصفح على عكس الفصول السابقة، فهي ليست جافاسكربت خام وليست مكتوبةً للمتصفح وإنما مكتوبة من أجل Node، فإذا أردت تشغيل هذه الشيفرات فستحتاج إلى تثبيت Node.js الإصدار 10.1 أو الأحدث بالذهاب إلى الموقع https://nodejs.org واتباع إرشادات التثبيت لنظام تشغيلك، كما ستجد هناك توثيقًا أكثر تفصيلًا عن Node.js.

20.1 تقديم إلى Node

تُعَدّ إدارة الدخل والخرج إحدى المشكلات الصعبة في كتابة أنظمة تتواصل عبر الشبكة، أي قراءة وكتابة البيانات من وإلى الشبكة والأقراص الصلبة، ذلك أنّ نقل البيانات من مكان لآخر يستغرق وقتًا؛ أما إذا جدولنا ذلك النقل فيمكن إحداث فرق كبير في سرعة استجابة النظام إلى طلبات المستخدِم أو الشبكة، كما تكون

البرمجة غير المتزامنة asynchronous مفيدةً في مثل تلك الحالات، فهي تسمح للبرنامج بإرسال واستقبال بيانات من وإلى أجهزة متعددة في الوقت نفسه دون إدارة معقدة للخيوط والتزامن.

وُضع تصور Node في البداية من أجل تسهيل البرمجة غير المتزامنة، كما تتكامل جافاسكربت جيدًا مع Node، فهي إحدى لغات البرمجة القليلة التي ليس لديها طريقة مضمَّنة لتنفيذ الدخل والخرج، وبالتالي يمكن استخدام جافاسكربت مع منظور Node غير المركزي للدخل والخرج دون أن يكون لدينا واجهتين غير متناسقتين، وقد نفَّذ الناس البرمجة المبنية على الاستدعاءات الخلفية في المتصفح في عام 2009 بالفعل حين صُمِّمت Node، وعليه فقد كان المجتمع الذي عاصر هذه اللغة معتادًا على تنسيق البرمجة غير المتزامن.

20.2 الأمر node

توفِّر Node.js عند تثبيتها على النظام برنامجًا اسمه node يُستخدَم لتشغيل ملفات جافاسكربت، فلنقل مثلًا أنه لدينا ملفًا اسمه hello.js يحتوي الشيفرة التالية:

```
let message = "Hello world";
console.log(message);
```

نستطيع تشغيل node من سطر الأوامر كما يلي لتنفيذ البرنامج:

```
$ node hello.js
Hello world
```

ينفِّذ التابع console.log شيئًا شبيهًا بما يفعله في المتصفح أي سيطبع جزءًا من النص، لكن سيذهب النص في Node إلى مجرى الخرج القياسي للعملية بدلًا من منصة جافاسكربت التي في المتصفح، وهذا يعني أننا سنرى القيم المسجَّلة في طرفيتنا؛ أما إذا شغّلت node دون إعطائها ملفًا، فستزودك بمحث prompt تستطيع كتابة شيفرة جافاسكربت فيه وترى نتيجتها مباشرةً.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

تكون الرابطة process متاحةً عمومًا في Node شأنها في ذلك شأن الرابطة console، حيث توفِّر طرقًا مختلفةً لفحص وتعديل البرنامج الحالي؛ أما التابع exit فينهي العملية ويمكن إعطائه رمز حالة خروج تخبر

البرنامج الذي بدأ node -وهي صدفية سطر الأوامر في هذه الحالة- هل اكتمل البرنامج بنجاح - أي الرمز صفر- أم قابل خطأ -أي رمز آخر-.

تستطيع قراءة process.argv لإيجاد الوسائط التي أُعطيت للسكربت الخاصة بك والتي هي مصفوفة من سلاسل نصية، لاحظ أنها تتضمن أمر node نفسه واسم السكربت الخاص بك، وبالتالي تبدأ الوسائط الحقيقية عند الفهرس 2، فإذا احتوى showargv.js على التعليمة console.log(process.argv)، فستستطيع تشغيلها كما يلي:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

توجد جميع رابطات جافاسكربت العامة مثل Array وMath وJSON في بيئة Node على عكس الوظائف المتعلِّقة بالمتصفح مثل document وprompt.

20.3 الوحدات Modules

تضيف Node بعض الرابطات الإضافية في النطاق العام global scope على الرابطات التي ذكرناها قبل قليل، فإذا أردت الوصول إلى الوظائف المضمَّنة، فيمكنك طلب ذلك من نظام الوحدات module system، وقد ذكرنا نظام وحدات CommonJS المبني على دالة require في الفصل العاشر؛ أما هذا النظام فقد دُمِج في Node ويُستخدَم لتحميل أيّ شيء بدءًا من الوحدات المضمَّنة إلى الحزم المحمَّلة إلى الملفات التي هي جزء من برنامجك.

يجب أن تحل Node السلسلة النصية المعطاة إلى ملف حقيقي يمكن تحميله عند استدعاء require، كما تُحَل أسماء المسارات التي تبدأ بـ / و./ و../ نسبيًا إلى مسار الوحدة الحالية؛ أما . فتشير إلى المجلد الحالي وتشير ../ إلى مجلد واحد للأعلى وتشير / إلى جذر نظام الملفات، فإذا طلبنا "./graph" من الملف /tmp/robot/robot.js، فستحاول Node تحميل الملف /tmp/robot/graph.js.

يمكن إهمال الامتداد .js، كما ستضيفه Node تلقائيًا إذا وُجد ملف بذلك الاسم، فإذا أشار المسار المطلوب إلى مجلد، فستحاول Node تحميل الملف الذي يكون اسمه index.js في ذلك المجلد، وإذا أعطينا سلسلةً نصيةً لا تبدو مسارًا نسبيًا أو مطلقًا إلى الدالة require، فستفترض أنها تشير إما إلى وحدة مضمَّنة أو وحدة مثبَّتة في المجلد node_modules، كما تعطينا require("fs") مثلًا وحدة نظام الملفات المضمَّن في Node؛ أما require("robot") فستحاول تحميل المكتبة الموجودة في node_modules/robot/، ويمكن تثبيت مثل تلك المكتبات باستخدام NPM الذي سنعود إليه بعد قليل.

لنعدّ الآن مشروعًا صغيرًا يتكون من ملفين، الأول اسمه main.js بحيث يعرّف سكربت يمكن استدعاؤها من سطر الأوامر لعكس سلسلة نصية.

```
const {reverse} = require("./reverse");

// Index 2 holds the first actual command line argument
let argument = process.argv[2];

console.log(reverse(argument));
```

يعرِّف الملف reverse.js مكتبةً لعكس السلاسل النصية التي يمكن استخدامها بواسطة أداة سطر الأوامر هذه وكذلك بواسطة السكربتات الأخرى التي تحتاج إلى وصول مباشر إلى دالة عكس سلاسل نصية.

```
  exports.reverse = function(string) {
return Array.from(string).reverse().join("");
  };
```

تذكَّر أنّ إضافة الخصائص إلى exports يضيفها إلى واجهة الوحدة، وبما أنّ Node.js تعامل الملفات على أساس وحدات CommonJS، فيمكن أن تأخذ main.js دالة reverse المصدَّرة من reverse.js، ونستطيع الآن استدعاء أداتنا كما يلي:

```
$ node main.js JavaScript
tpircSavaJ
```

20.4 التثبيت باستخدام NPM

تعرَّضنا إلى مستودع NPM في الفصل العاشر وهو مستودع لوحدات جافاسكربت، وقد كُتب الكثير منها من أجل Node، فإذا ثبَّت Node على حاسوبك، فستستطيع الحصول على أمر npm الذي يمكن استخدامه للتفاعل مع ذلك المستودع، والغرض الأساسي من NPM هو تحميل الحِزم، حيث نستطيع استخدامه لجلب وتثبيت حزمة ini التي رأيناها في الفصل العاشر على حاسوبنا:

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
 open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s

$ node
```

```
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

يجب أن ينشئ NPM مجلدًا اسمه node_modules بعد تشغيل npm install، حيث سيكون مجلد ini بداخل ذلك المجلد محتويًا على المكتبة التي يمكن فتحها والاطلاع على شيفرتها، وتُحمَّل تلك المكتبة عند استدعاء require("ini")، كما نستطيع استدعاء الخاصية parse الخاصة بها لتحليل ملف الإعدادات.

يثبِّت NPM الحِزم تحت المجلد الحالي افتراضيًا بدلًا من المكان المركزي، وقد يكون هذا غريبًا إذا كنا معتادين على مدير حِزم آخر، لكن هذا له مزاياه، فهو يجعل كل تطبيق متحكمًا بالكامل في الحِزم التي يثبِّتها ويسهِّل إدارة الإصدارات ومحو التطبيقات إذا أردنا حذفها.

20.5 ملفات الحزم

تستطيع رؤية تحذير في مثال npm install أنّ الملف package.json غير موجود، كما يُنصح بإنشاء مثل هذا الملف لكل مشروع إما يدويًا أو عبر تشغيل npm init، حيث يحتوي على بعض معلومات المشروع مثل اسمه وإصداره ويسرد اعتمادياته، ولنضرب مثلًا هنا بمحاكاة الروبوت من الفصل السابع التي عدلنا عليها عند تعرضنا للوحدات في الفصل العاشر، إذ قد يبدو ملف package.json الخاص بها كما يلي:

```
  {
"author": "Marijn Haverbeke",
"name": "eloquent-JavaScript-robot",
"description": "Simulation of a package-delivery robot",
"version": "1.0.0",
"main": "run.js",
"dependencies": {
  "dijkstrajs": "^1.0.1",
  "random-item": "^1.0.0"
},
"license": "ISC"
  }
```

عند تشغيل npm install دون تسمية الحزمة المراد تثبيتها، فسيثبِّت NPM الاعتماديات التي يسردها package.json؛ أما إذا ثبَّتت حزمةً ما ليست موجودة على أساس اعتمادية، فسيضيفها NPM إلى package.json.

20.6 الإصدارات

يسرد ملف package.json كلًا من إصدار البرنامج وإصدارات اعتمادياته، والإصدارات هي أسلوب للتعامل مع التطور المنفصل للحِزم، فقد لا تعمل الشيفرة التي كُتبت لحزمة في وقت ما مع الإصدار الجديد والمعدل من تلك الحزمة، حيث يشترط NPM أن تتبع الحِزم الخاصة به نظامًا اسمه الإصدار الدلالي semantic versioning الذي يرمّز بعض المعلومات عن الإصدارات المتوافقة التي لا تعطل الواجهة القديمة في رقم الإصدار version number، حيث يتكون الإصدار الدلالي من ثلاثة أعداد مفصولة بنقاط مثل 2.3.0، وكل مرة تضاف فيها ميزة جديدة فإننا نزيد العدد الأوسط، وكل مرة تُعطل فيها التوافقية بحيث لا تعمل الشيفرة الحالية التي تستخدِم الحزمة مع إصدارها الجديد فإننا نغير العدد الأول من اليسار.

يوضِّح محرف الإقحام ^ الذي يكون أمام رقم إصدار الاعتمادية في package.json أنّ أيّ نسخة متوافقة مع الرقم المعطى يمكن تثبيتها، وعلى ذلك تعني "^2.3.0" أنّ أيّ نسخة أكبر من أو يساوي 2.3.0 وأقل من 3.0.0 مسموح بها، كما يُستخدَم أمر npm أيضًا لنشر حزم جديدة أو إصدارات جديدة من الحزم، فإذا شغّلنا npm publish في مجلد فيه ملف package.json، فسينشر حزمةً بالاسم والإصدار الموجودَين في ملف JSON إلى السجل registry، ويستطيع أيّ أحد نشر حزم في NPM لكن شرط أن يكون اسم الحزمة غير مستخدَم من قبل.

ليس ثمة شيء فريد في وظيفته بما أنّ برنامج npm جزء برمجي يتواصل مع نظام مفتوح هو سجل الحزم، ويمكن تثبيت برنامج آخر مثل yarn من سجل NPM ليؤدي وظيفة npm نفسها باستخدام واجهة مختلفة قليلًا وكذلك استراتيجية تثبيت مختلفة، لكن هذا الكتاب لا يتعمق في استخدام NPM، وإنما ننصحك بالرجوع إلى https://npmjs.org لمزيد من التوثيق والبحث عن الحزم.

20.7 وحدة نظام الملفات

إحدى الوحدات المضمَّنة والمستخدَمة بكثرة في Node هي وحدة fs التي تشير إلى نظام الملفات file system، إذ تصدِّر الدوال من أجل العمل مع الملفات والمجلدات، حيث تقرأ مثلًا الدالة readFile ملفًا وتستدعي رد نداء بمحتويات الملف كما يلي:

```
  let {readFile} = require("fs");
  readFile("file.txt", "utf8", (error, text) => {
if (error) throw error;
console.log("The file contains:", text);
  });
```

يشير الوسيط الثاني لدالة readFile إلى ترميز المحارف المستخدَم لفك تشفير الملف إلى سلسلة نصية، ورغم وجود عدة طرق لتشفير النصوص إلى بيانات ثنائية إلا أنّ أغلب النظم الحديثة تستخدِم UTF-8، فإذا لم

يكن لديك سبب يجعلك تفضل ترميزًا آخر غير هذا فإنك تستخدمه، وعليه تمرر "utf8" عند قراءة ملف نصي، فإذا لم تمرر ترميزًا، فستفترض Node أنك تريد البيانات الثنائية وستعطيك الكائن Buffer بدلًا من سلسلة نصية، وهو كائن شبيه بالمصفوفة يحتوي أعدادًا تمثل البايتات (قطع بيانات بحجم 8 بت) التي في الملف.

```
  const {readFile} = require("fs");
  readFile("file.txt", (error, buffer) => {
if (error) throw error;
console.log("The file contained", buffer.length, "bytes.",
"The first byte is:", buffer[0]);
  });
```

تُستخدَم الدالة writeFile لكتابة ملف إلى القرص الصلب كما يلي:

```
  const {writeFile} = require("fs");
  writeFile("graffiti.txt", "Node was here", err => {
if (err) console.log(`Failed to write file: ${err}`);
else console.log("File written.");
  });
```

ليس من الضروري هنا تحديد الترميز، إذ ستفترض الدالة عند إعطائها سلسلة نصية أنّ عليها كتابتها على أساس نص باستخدام ترميزها الافتراضي للمحارف -أي UTF-8- ما لم يكن كائن Buffer.

تحتوي الوحدة fs على عدة دوال أخرى مفيدة مثل readdir التي ستعيد الملفات الموجودة في مجلد على أساس مصفوفة من السلاسل النصية، وstat التي ستجلب معلومات عن ملف ما وrename التي ستعيد تسمية الملف وunlink التي ستحذِف الملفات وهكذا، ولمزيد من التفاصيل انظر توثيق Node، كما تأخذ أغلب الدوال السابقة دالة رد نداء على أساس آخر معامِل لها وتستدعيها إما مع خطأ -أي أول وسيط- أو مع نتيجة ناجحة -أي ثاني وسيط-، ورأينا في الفصل الحادي عشر وجود تبعات لمثل هذا التنسيق من البرمجة لعل أكبرها أن عملية معالجة الأخطاء نفسها تصبح طويلة وعرضة للخطأ.

لا زال تكامل الوعود مع Node قيد التطوير وقت كتابة هذه الكلمات رغم أنها أُدخلت في جافاسكربت منذ مدة لا بأس بها، فلدينا الكائن promises المصدَّر من حزمة fs منذ الإصدار 10.1 والذي يحتوي على أغلب الدوال الموجودة في fs لكنه يستخدِم الوعود بدلًا من دوال رد النداء.

```
  const {readFile} = require("fs").promises;
  readFile("file.txt", "utf8")
.then(text => console.log("The file contains:", text));
```

قد لا نحتاج أحيانًا إلى اللاتزامنية، بل قد تعيق عملنا، ولحسن حظنا أنّ أغلب الدوال التي في fs نسخة تزامنية لها الاسم نفسه مع لاحقة Sync مضافة إلى آخرها، فسيكون اسم النسخة التزامنية من دالة readFile مثلًا readFileSync.

```
const {readFileSync} = require("fs");
console.log("The file contains:",
readFileSync("file.txt", "utf8"));
```

لاحظ أنّ البرنامج سيتوقف عن العمل تمامًا أنه ريثما تُنفذ العملية التزامنية، وسيكون ذلك تأخيرًا مزعجًا إذا كان يُفترض به الاستجابة إلى المستخدِم أو إلى آلات أخرى في الشبكة أثناء تلك العملية.

20.8 وحدة HTTP

لدينا وحدة مركزية أخرى توفِّر وظيفة تشغيل خوادم HTTP وإنشاء طلبات HTTP كذلك واسمها http، وإذا أردنا تشغيل خادم HTTP فسيكون ذلك عبر السكربت التالية:

```
  const {createServer} = require("http");
  let server = createServer((request, response) => {
response.writeHead(200, {"Content-Type": "text/html"});
response.write(`
  <h1>Hello!</h1>
  <p>You asked for <code>${request.url}</code></p>`);
response.end();
  });
  server.listen(8000);
  console.log("Listening! (port 8000)");
```

فإذا شغّلت هذه السكربت على الحاسوب، فستوجِّه المتصفح إلى http://localhost:8000/hello لإنشاء طلب إلى خادمك، وسيستجيب بصفحة HTML صغيرة، كما تُستدعى الدالة الممررة على أساس وسيط إلى createServer في كل مرة يتصل عميل بالخادم، كذلك تُعَدّ الرابطتان request وresponse كائنين يمثلان البيانات الواردة والصادرة، حيث يحتوي الأول على معلومات عن الطلب مثل خاصية url الخاصة به والتي تخبرنا بالرابط URL الذي أُنشئ الطلب إليه، لذلك عندما نفتح تلك الصفحة في المتصفح فإنها ترسل طلبًا إلى حاسوبك، وهذا يشغّل دالة الخادم ويرجع استجابةً نراها في المتصفح.

أما لإرجاع شيء من طرفنا فستستدعي عدة توابع على الكائن response، أولها هو التابع writeHead الذي يكتب ترويسات الاستجابة -انظر الفصل الثامن عشر-، وتعطيه شيفرة الحالة -أي 200 التي تعني OK في هذه الحالة-، وكائنًا يحتوي على قيم الترويسة، ثم يعيِّن المثال ترويسة Content-Type لتخبر العميل أننا نرسل

مستند HTML إليه، ثم يُرسَل متن الاستجابة الفعلية -أي المستند نفسه- باستخدام `response.write`، ويُسمح لنا باستدعاء هذا التابع عدة مرات إذا أردنا إرسال الاستجابة جزءًا جزءًا، كما في حالة بث البيانات إلى العميل كلما صارت متاحةً على سبيل المثال، ثم تشير `response.end` إلى نهاية الاستجابة.

يتسبب استدعاء `server.listen` في جعل الخادم ينتظر اتصالًا على المنفَذ 8000 وهذا هو السبب الذي يجبرنا على الاتصال بـ localhost:8000 من أجل التواصل مع هذا الخادم بدلًا من localhost فقط والتي ستستخدِم المنفَذ 80، وتظل العملية في وضع انتظار عند تشغيل هذه السكربت، ولن تخرج `node` تلقائيًا عندما تصل إلى نهاية السكربت بما أنها تظل في وضع استماع إلى الإحداث وانتظارها والتي هي اتصالات الشبكة في هذه الحالة، كما نضغط control+c من أجل إغلاقها، وكان هذا الخادم مثالًا فقط، وإلا فإنّ خادم الويب الحقيقي يفعل أكثر من ذلك، فهو ينظر في تابع الطلب -أي الخاصية `method`- ليرى الإجراء الذي يحاول العميل تنفيذه، وينظر في رابط الطلب كي يعرف المورد الذي ينفَّذ عليه ذلك الإجراء، وسنرى لاحقًا في هذا الفصل خادمًا أكثر تقدمًا وتعقيدًا.

يمكننا استخدام الدالة `request` في وحدة `http` من أجل التصرف على أساس عميل HTTP.

```
  const {request} = require("http");
  let requestStream = request({
hostname: "eloquentJavaScript.net",
path: "/20_node.html",
method: "GET",
headers: {Accept: "text/html"}
  }, response => {
console.log("Server responded with status code",
response.statusCode);
  });
  requestStream.end();
```

يهيئ الوسيط الأول للدالة `request` الطلب ليخبر Node بالخادم الذي يجب التواصل معه والمسار الذي تطلبه من ذلك الخادم وأيّ تابع يجب استخدامه وهكذا؛ أما الوسيط الثاني فيكون الدالة التي يجب أن تستدعَى عندما تأتي الاستجابة، وتعطى كائنًا يسمح لنا بفحص الاستجابة، لمعرفة رمز حالتها مثلًا، كما يسمح الكائن الذي تعيده `request` ببث البيانات في الطلب باستخدام التابع `write` كما في كائن `response` الذي رأيناه في الخادم وتنهي الطلب بالتابع `end`، ولا يستخدم المثال `write` لأن طلبات `GET` يجب ألا تحتوي على بيانات في متونها.

لدينا دالة `request` مشابهةً في وحدة `https`، حيث يمكن استخدامها لإنشاء طلبات إلى روابط `https:`، ولا شك أنّ إنشاء الطلبات باستخدام Node الخام أمر طويل مسهب، لهذا توجد حزم تغليف

سهلة الاستخدام متاحة في NPM مثل node-fetch التي توفر واجهة fetch مبنيةً على الوعود التي عرفناها من المتصفح.

20.9 البث Stream

رأينا نسختين من البث القابل للكتابة writable stream في مثالَي HTTP، هما كائن الاستجابة الذي يستطيع الخادم كتابته، وكائن الطلب الذي أعادته request، حيث يُستخدَم البث القابل للكتابة كثيرًا في Node، فمثل تلك الكائنات لها تابع اسمه write يُمكن تمرير سلسلة نصية إليه، أو كائن Buffer لكتابة شيء في البث؛ أما التابع end فيغلق البث ويأخذ قيمةً -بصورة اختيارية- للكتابة في البث قبل الإغلاق، ويمكن إعطاء كلا التابعَين السابقين رد نداء على أساس وسيط إضافي، حيث يستدعيانه عند انتهاء الكتابة أو الإغلاق.

يمكن إنشاء بث قابل للكتابة يشير إلى ملف باستخدام دالة createWriteStream من وحدة fs، ثم يمكنك استخدام التابع write على الكائن الناتج من أجل كتابة الملف جزءًا واحدًا في كل مرة بدلًا من كتابته على مرة واحدة كما في writeFile؛ أما البث القابل للقراءة ففيه تفصيل أكثر، فرابطة request التي مُرِّرت إلى الاستدعاء الخلفي لخادم HTTP قابلة للقراءة، وكذلك رابطة response الممررة إلى رد نداء العميل HTTP. حيث يقرأ الخادم الطلبات ثم يكتب الاستجابات، بينما يكتب العميل الطلب أولًا ثم يقرأ الاستجابة، وتتم القراءة من البث باستخدام معالجات الأحداث بدلًا من التوابع.

تملك الكائنات التي تطلق الأحداث في Node تابعًا اسمه on يشبه التابع addEventListener الموجود في المتصفح، كما يمكن إعطاؤه اسم حدث ثم دالة، وسيسجل تلك الدالة لتُستدعى كلما وقع ذلك الحدث، كذلك فإن البث القابل للقراءة له حدثان هما "data" و"end"، حيث يُطلَق الأول في كل مرة تأتي بيانات فيها؛ أما الثاني فيُستدعى كلما كان البث عند نهايته، وهذا النموذج مناسب لبث البيانات التي يمكن معالجتها فورًا حتى لو كان باقي المستند غير متاح بعد، كما يُقرأ الملف على أساس بث قابل للقراءة من خلال استخدام دالة createReadStream من وحدة fs، وتنشئ الشيفرة التالية خادمًا يقرأ متون الطلبات ويبثها مرةً أخرى إلى العميل على أساس نص حروفه من الحالة الكبيرة:

```
  const {createServer} = require("http");
  createServer((request, response) => {
response.writeHead(200, {"Content-Type": "text/plain"});
request.on("data", chunk =>
  response.write(chunk.toString().toUpperCase()));
request.on("end", () => response.end());
  }).listen(8000);
```

ستكون القيمة chunk الممررة إلى معالج البيانات على هيئة Buffer ثنائي، ويمكن تحويل ذلك إلى سلسلة نصية بفك تشفيرها على أساس محارف UTF-8 مرمزة باستخدام التابع toString، منا ترسب الشيفرة التالية عند تشغيلها أثناء نشاط خادم الحروف الكبيرة طلبًا إلى ذلك الخادم وتكتب الاستجابة التي تحصل عليها:

```
  const {request} = require("http");
  request({
hostname: "localhost",
port: 8000,
method: "POST"
  }, response => {
response.on("data", chunk =>
  process.stdout.write(chunk.toString()));
  }).end("Hello server");
  // → HELLO SERVER
```

يكتب المثال إلى process.stout والذي يُعَدّ خرجًا قياسيًا للعملية وبثًا قابلًا للكتابة، بدلًا من استخدام console.log، إذ لا نستطيع استخدام console.log لأنها تضيف محرف سطر جديد إضافي بعد كل جزء تكتبه من النص، وهذا ليس مناسبًا هنا بما أنّ الاستجابة قد تأتي في هيئة عدة كتل نصية.

20.10 خادم الملفات

نريد الآن دمج المعلومات التي عرفناها عن خوادم HTTP والعمل مع نظام الملفات لإنشاء جسر بين خادم HTTP يسمح بالوصول البعيد إلى نظام ملفات، كما يكون في مثل تلك الخوادم جميع أنواع الاستخدامات الممكنة، فهي تسمح لتطبيقات الويب بتخزين البيانات ومشاركتها، أو تعطي مجموعةً من الناس وصولًا مشتركًا إلى مجموعة ملفات، ويمكن استخدام التوابع GET وPUT وDELETE لقراءة وكتابة وحذف الملفات على الترتيب، وذلك عندما نعامل الملفات على أساس موارد HTTP، كما سنفسر المسار الذي في الطلب على أنه مسار الملف الذي يشير إليه الطلب، ولعلنا لا نريد مشاركة كل نظام الملفات الخاص بنا، لذا سنفسر تلك المسارات على أنها تبدأ في المجلد العامل للخادم وهو المجلد الذي بدأ فيه.

فإذا شغّلنا الخادم من /tmp/public/ أو على ويندوز من C:\tmp\public\، فسيشير طلب /file.txt إلى /tmp/public/file.txt أو C:\tmp\public\file.txt على ويندوز، كما سنبني البرنامج جزءًا جزءًا مستخدِمين الكائن methods لتخزين الدوال التي تعالج توابع HTTP المختلفة، وتكون معالجات التوابع دوال async تحصل على كائن الطلب على أساس وسيط وتعيد وعدًا يُحل إلى كائن يصف الاستجابة.

```
  const {createServer} = require("http");
  const methods = Object.create(null);

    createServer((request, response) => {
let handler = methods[request.method] || notAllowed;
handler(request)
  .catch(error => {
if (error.status != null) return error;
return {body: String(error), status: 500};
  })
  .then(({body, status = 200, type = "text/plain"}) => {
 response.writeHead(status, {"Content-Type": type});
 if (body && body.pipe) body.pipe(response);
 else response.end(body);
  });
  }).listen(8000);

  async function notAllowed(request) {
return {
  status: 405,
  body: `Method ${request.method} not allowed.`
};
  }
```

يبدأ هذا خادمًا لا يعيد إلا استجابات خطأ 405، وهو الرمز الذي يشير إلى رفض الخادم لمعالجة التابع المعطى.

يحوِّل استدعاء `catch` عند رفض وعد معالِج الطلب الخطأ إلى كائن استجابة إذا لم يكن هو كائن استجابة بالفعل، وذلك كي يستطيع الخادم إرسال استجابة خطأ مرةً أخرى لإخبار العميل أنه فشل في معالجة الطلب، كما يمكن إهمال حقل `status` في وصف الاستجابة وتكون حينئذ 200 افتراضيًا -وهي التي تعني OK-؛ أما نوع المحتوى في الخاصية `type` فيمكن إهماله كذلك، ويفترض حينها أنّ الاستجابة نص مجرد.

حين تكون قيمة `body` بثًا قابلًا للقراءة فسيحتوي على التابع `pipe` الذي يُستخدَم لإعادة توجيه كل المحتوى من بث قابل للقراءة إلى بث قابل للكتابة، وإلا فيُفترض أنه إما `null` -أي لا شيء- أو سلسلةً نصيةً أو مخزنًا مؤقتًا buffer، ويُمرَّر مباشرة إلى التابع `end` الخاص بالاستجابة، كما تستخدِم الدالة `urlPath` وحدة `url` المضمَّنة لتحليل الرابط من أجل معرفة مسار الملف المتوافق مع رابط الطلب، وهي تأخذ اسم المسار الذي

يكون شيئًا مثل `"/file.txt"` وتفك تشفيره لتتخلص من رموز التهريب التي على شاكلة `%20` وتحله نسبة إلى المجلد العامل للبرنامج.

```
  const {parse} = require("url");
  const {resolve, sep} = require("path");

  const baseDirectory = process.cwd();

  function urlPath(url) {
let {pathname} = parse(url);
let path = resolve(decodeURIComponent(pathname).slice(1));
if (path != baseDirectory &&
!path.startsWith(baseDirectory + sep)) {
  throw {status: 403, body: "Forbidden"};
}
return path;
  }
```

يجب عليك القلق بشأن الأمان عند تهيئة وضبط البرنامج ليقبل طلبات الشبكة، ففي حالتنا من الممكن كشف كل نظام الملفات إلى الشبكة إذا لم نتوخَّ الحذر.

تُعَدّ مسارات الملفات سلاسل نصية في Node، حيث يلزمنا قدر لا بأس به من التفسير interpretation لربط مثل تلك السلسلة النصية بملف حقيقي، فقد تحتوي المسارات على `../` لتشير إلى المجلد الأب، وعليه يكون أحد المصادر البدهية للمشكلة هي طلبات إلى مسارات مثل `/../secret_file`، ومن أجل تجنب مثل تلك المشاكل تستخدِم `urlPath` الدالة `resolve` من وحدة `path` التي تحل الروابط النسبية، ثم تتحقق من كون النتيجة تحت المجلد العامل دائمًا، كما يمكن استخدام الدالة `process.cwd` لإيجاد ذلك المجلد العامل، حيث تشير cwd إلى المجلد العامل الحالي أو current working directory.

أما رابطة `sep` من حزمة `path` فهي فاصلة مسار النظام، وهي شرطة مائلة خلفية على ويندوز وأمامية على أغلب نظم التشغيل الأخرى، فإذا لم يبدأ المسار بالمجلد الرئيسي فسترفع الدالة كائن استجابة خطأ باستخدام رمز حالة HTTP يشير إلى استحالة الوصول إلى المورد، وسنضبط التابع `GET` كي يعيد قائمةً من الملفات عند قراءة مجلد ويعيد محتوى الملف عند قراءة ملف عادي؛ أما السؤال الذي يطرح نفسه هنا هو نوع ترويسة `Content-Type` التي يجب تعيينها عند إعادة محتوى الملف، فبما أن تلك الملفات قد تكون أيّ شيء فلا يستطيع الخادم إعادة نوع المحتوى نفسه في كل مرة، ونستخدِم هنا NPM مرةً أخرى، إذ تعرف حزمة `mime` النوع الصحيح لعدد كبير من امتدادات الملفات، كما تسمى موضحات أنواع المحتوى مثل `text/plain` باسم mime، يثبّت الأمر `npm` أدناه إصدارًا محددًا من `mime` في المجلد الذي فيه سكربت الخادم:

```
$ npm install mime@2.2.0
```

يعاد رمز الحالة 404 إذا لم يكن الملف المطلوب موجودًا، وسنستخدم الدالة `stat` التي تبحث عن معلومات تتعلق بملف ما لتعرف هل الملف موجود أم لا وهل هو مجلد أم لا.

```
  const {createReadStream} = require("fs");
  const {stat, readdir} = require("fs").promises;
  const mime = require("mime");

  methods.GET = async function(request) {
let path = urlPath(request.url);
let stats;
try {
  stats = await stat(path);
} catch (error) {
  if (error.code != "ENOENT") throw error;
  else return {status: 404, body: "File not found"};
}
if (stats.isDirectory()) {
  return {body: (await readdir(path)).join("\n")};
} else {
  return {body: createReadStream(path),
  type: mime.getType(path)};
}
  };
```

تكون `stat` لا تزامنيةً لأنها ستحتاج أن تتعامل مع القرص وستأخذ وقتًا لذلك، وبما أننا نستخدِم الوعود بدلًا من تنسيق رد النداء فيجب استيراده من `promises` بدلًا من `fs` مباشرةً، حيث ترفع `stat` كائن خطأ به الخاصية `code` لـ `"ENOENT"` إذا لم يكن الملف موجودًا، وإذا بدت هذه الرموز غريبةً عليك لأول وهلة فاعلم أنها متأثرة بأسلوب نظام يونكس، كما ستجد أنواع الخطأ في Node على مثل هذه الشاكلة.

يخبرنا الكائن `stats` الذي تعيده `stat` بمعلومات عديدة عن الملف مثل حجمه -أي الخاصية `size`- وتاريخ التعديل عليه -أي الخاصية `mtime`- وهل هذا الملف مجلد أم ملف عادي من خلال التابع `isDirectory`، كما نستخدِم `readdir` لقراءة مصفوفة ملفات في المجلد ونعيدها إلى العميل؛ أما بالنسبة للملفات العادية فسننشئ بثًا قابلًا للقراءة باستخدام `createReadStream` ونعيده على أنه المتن مع نوع المحتوى الذي تعطينا إياه الحزمة `mime` لاسم الملف، كما تكون الشيفرة التي تعالج طلبات `DELETE` أبسط قليلًا.

```
  const {rmdir, unlink} = require("fs").promises;

  methods.DELETE = async function(request) {
let path = urlPath(request.url);
let stats;
try {
  stats = await stat(path);
} catch (error) {
  if (error.code != "ENOENT") throw error;
  else return {status: 204};
}
if (stats.isDirectory()) await rmdir(path);
else await unlink(path);
return {status: 204};
  };
```

إذا لم تحتوي استجابة HTTP على أيّ بيانات فيمكن استخدام رمز الحالة 204 ("لا محتوى") لتوضيح ذلك، وهو الخيار المنطقي هنا بما أنّ الاستجابة للحذف لا تحتاج أيّ معلومات أكثر من تأكيد نجاح العملية، لكن لماذا نحصل على رمز حالة يفيد النجاح عند محاولة حذف ملف غير موجود أصلًا؟ أليس من المنطقي أن نحصل على خطأ؟

يرجع ذلك إلى معيار HTTP الذي يشجعنا على جعل الطلبات راسخة idempotent، مما يعني سيعطينا تكرار الطلب نفسه النتيجة نفسها التي خرجت في أول مرة، فإذا حاولنا حذف شيء ليس موجودًا فيمكن القول أنّ التأثير الذي كنا نحاول إحداثه قد وقع -وهو فعل الحذف-، فلم يَعد العنصر الذي نريد حذفه موجودًا، وكأن هدف الطلب قد تحقق كما لو كان موجودًا ثم حذفناه بطلبنا، كما تمثِّل الشيفرة التالية معالِج طلبات PUT:

```
  const {createWriteStream} = require("fs");

  function pipeStream(from, to) {
return new Promise((resolve, reject) => {
  from.on("error", reject);
  to.on("error", reject);
  to.on("finish", resolve);
  from.pipe(to);
});
  }
```

```
  methods.PUT = async function(request) {
let path = urlPath(request.url);
await pipeStream(request, createWriteStream(path));
return {status: 204};
  };
```

لسنا في حاجة إلى التحقق من وجود الملف هذه المرة، فإذا كان موجودًا فسنكتب فوقه، ونستخدِم `pipe` هنا مرةً أخرى لنقل البيانات من البث القابل للقراءة إلى بث قابل للكتابة، وفي حالتنا هذه من الطلب إلى الملف، لكن بما أنّ `pipe` ليست مكتوبةً لتعيد وعدًا، فعلينا كتابة مغلِّف هو `pipeStream` الذي ينشئ وعدًا حول ناتج استدعاء `pipe`، كما سيعيد `createWriteStream` بثًا إذا حدث خطأ أثناء فتح الملف لكن سيطلق ذلك البث حدث `"error"`، وقد يفشل البث من الطلب كما في حالة انقطاع الشبكة، لذا فإننا نوصل الحدثين `"error"` لكلا البثين كي يرفضا الوعد.

سيغلق بث الخرج الذي يتسبب في إطلاق الحدث `"finish"` عند انتهاء `pipe`، وهي النقطة التي يمكننا حل الوعد فيها بنجاح -أي لا نعيد شيئًا-، كما يمكن العثور على السكربت الكاملة للخادم في https://eloquentJavaScript.net/code/file_server.js وهي متاحة للتحميل، وتستطيع بدء خادم الملفات الخاص بك بتحميلها وتثبيت اعتمادياتها ثم تشغيلها مع Node، كما تستطيع تعديلها وتوسيعها لحل تدريبات هذا الفصل أو للتجربة، وتُستخدم أداة سطر الأوامر `curl` لإنشاء طلبات HTTP، وهي أداة متاحة في الأنظمة الشبيهة بنظام يونكس UNIX مثل ماك ولينكس وما شابههما، كما تختبر الشيفرة التالية خادمنا، حيث تستخدِم الخيار `-X` لتعيين تابع الطلب و`-d` لإدراج متن الطلب.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

يفشل الطلب الأول إلى `file.txt` لأنّ الملف غير موجود بعد، لكن الطلب الثاني ينجح في جلب الملف بعد إنشاء طلب `PUT` لذلك الملف، ثم بعد ذلك يُفقد الملف مرةً أخرى بسبب طلب `DELETE` الذي يحذفه.

20.11 خاتمة

تسمح منصة Node لنا بتشغيل جافاسكربت في سياق خارج المتصفح، وقد صُممت أساسًا من أجل مهام الشبكات لتلعب دور عقدة -كما يشير الاسم Node- داخل شبكة ما، لكنها تتكامل جيدًا مع مهام السكربتات باختلاف أنواعها، وستستمتع بأتمتة المهام بها إذا كنت تحب جافاسكربت، كما يوفِّر NPM حزمًا لكل شيء تقريبًا ويسمح لنا بجلب وتثبيت تلك الحزم باستخدام البرنامج `npm`، كما تأتي Node بعدد من الوحدات المضمَّنة مثل وحدة `fs` التي تعمل مع نظام الملفات ووحدة `http` التي تشغِّل خوادم HTTP وتنشئ طلبات HTTP أيضًا.

تُنفَّذ جميع عمليات الإدخال والإخراج في Node بأسلوب غير متزامن إلا إذا استخدَمت نسخةً متزامنةً من الدالة صراحةً مثل `readFileSync`، كما يجب توفر دوال رد نداء عند استدعاء مثل تلك الدوال غير المتزامنة، وستستدعيها Node بقيمة خاطئة ونتيجة إذا كانت جاهزةً ومتاحةً.

20.12 تدريبات

20.12.1 أداة بحث

توجد أداة سطر أوامر في UNIX للبحث السريع في الملفات عن تعبير نمطي وهي أداة `grep`.

اكتب سكربت Node يمكن تشغيلها من سطر الأوامر وتتصرف مثل `grep`، بحيث تعامل أول وسيط سطر أوامر على أساس تعبير نمطي، وتعامل بقية الوسائط على أساس ملفات يجب البحث فيها، كما يجب أن يكون الخرج اسم الملف الذي يطابق محتواه التعبير النمطي، وإذا نجحت في هذا فوسِّع الأداة بحيث إذا كان أحد الوسائط مجلدًا فستبحث في جميع الملفات في ذلك المجلد ومجلداته الفرعية أيضًا.

استخدم دوال تزامنية أو لا تزامنية وفق ما تراه مناسبًا، فرغم أنّ إعداد السكربت بحيث يمكن طلب عدة إجراءات غير متزامنة في الوقت نفسه قد يسرع البحث قليلًا، لكن ليس بالقدر الذي يكون فارقًا عن النمط التزامني بما أنّ نظم الملفات لا تستطيع قراءة أكثر من شيء واحد في كل مرة.

إرشادات الحل

ستجد الوسيط الأول لك -وهو التعبير النمطي- في `process.argv[2]`، ثم تأتي ملفات الدخل بعد ذلك، ويمكنك استخدام الباني `RegExp` للتحويل من سلسلة نصية إلى كائن تعبير نمطي، ولا شك أنّ تنفيذ هذه السكربت تزامنيًا باستخدام `readFileSync` سيكون أبسط وأسهل، لكن إذا استخدمت `fs.promises` من أجل الحصول على دوال تعيد وعودًا وكتبت دالة `async`، فلن تبدو الشيفرة غريبةً أو مختلفةً، كما يمكنك استخدام `stat` أو `statSync` والتابع `isDirectory` الخاص بكائن `stat` لمعرفة هل العنصر المبحوث عنه مجلد أم لا.

تُعَدّ عملية تصفح مجلد عمليةً متفرعةً، حيث يمكنك تنفيذها باستخدام دالة تعاودية أو بالاحتفاظ بمصفوفة عمل -أي ملفات يجب تصفحها-.، كما تستطيع استدعاء `readdir` أو `readdirSync` للبحث عن ملفات في مجلد ما، وعليك ملاحظة أنّ أسلوب التسمية في دوال Node يختلف عن جافاسكربت وهو أقرب إلى أسلوب دوال يونكس القياسية، كما في `readdir` التي تكون كل الحروف فيها من الحالة الصغيرة، ثم نضيف `Sync` بحرف S كبير، وإذا أردت الذهاب من ملف قرأته `readdir` إلى الاسم الكامل للسمار، فيجب جمعه إلى اسم المجلد بوضع محرف شرطة مائلة `/` بينهما.

20.12.2 إنشاء المجلد

رغم استطاعة التابع `DELETE` الذي في خادم ملفاتنا حذف المجلدات باستخدام `rmdir` إلا أنّ الخادم لا يدعم حاليًا أي طريقة لإنشاء مجلد، لذا أضف دعمًا للتابع `MKCOL` -الذي يعني أنشئ تجميعةً Make Collection-، والذي سينشئ مجلدًا باستدعاء `mkdir` من وحدة `fs`.

لا يُستخدم `MKCOL` -وهو تابع HTTP- كثيرًا لكنه موجود لمثل هذا الغرض تحديدًا في معيار WebDAV الذي يحدِّد مجموعةً من الأساليب فوق HTTP لتجعله مناسبًا لإنشاء المستندات.

إرشادات الحل

يمكنك استخدام الدالة التي تستخدِم التابع `DELETE` على أساس نموذج للتابع `MKCOL`، وحاول إنشاء مجلد باستخدام `mkdir` إذا لم يُعثر على ملف؛ أما إذا وجد مجلد في ذلك المسار فأعد الاستجابة 204 كي تكون طلبات إنشاء المجلدات راسخةً idempotent، فإذا وجد ملف لا يكون مجلدًا هنا فأعد رسالة خطأ، وسيكون رمز الخطأ 400 -أي "طلب سيء bad request"- هو المناسب.

20.12.3 مساحة عامة على الويب

بما أن خادم الملفات يتعامل مع أيّ نوع من أنواع الملفات، بل ويدرِج ترويسة `Content-Type` المناسبة، فيمكنك استخدامه لخدمة موقع ما، كما سيكون موقعًا فريدًا بما أنه يسمح لأيّ أحد بحذف الملفات واستبدالها، حيث سيكون موقعًا يمكن تعديله وتحسينه وتخريبه كذلك من قِبل أيّ أحد لديه وقت لإنشاء طلبHTTP مناسب.

اكتب صفحة HTML تدرِج ملف جافاسكربت بسيط، وضَع الملفات في مجلد يستطيع خادم الملفات الوصول إليه ويخدمه وافتحها في المتصفح، ثم ابن واجهةً صديقةً للمستخدِم لتعديل الموقع من داخل الموقع نفسه مستفيدًا من المعلومات التي حصلتها في هذا الكتاب وعلى أساس تدريب متقدم قليلًا أو حتى على أساس مشروع لنهاية الأسبوع.

استخدم استمارة HTML لتعديل محتوى الملفات التي تكوّن الموقع بما يسمح للمستخدِم بتحديثها على الخادم من خلال استخدام طلبات HTTP كما ذكرنا في الفصل الثامن عشر، وابدء بجعل ملف واحد فقط قابلًا

للتعديل ثم أكثر من ملف بحيث يستطيع المستخدِم اختيار أيّ ملف يمكن تعديله، واستفد من كون خادم الملفات يعيد قائمةً من الملفات عند قراءة مجلد ما، كما لا تعمل في الشيفرة المعروضة لخادم الملفات مباشرةً بما أنك قد تخطئ فتدمِّر الملفات التي هناك، بل اجعل عملك خارج المجلد العام وانسخه عند الاختبار.

إرشادات الحل

تستطيع إنشاء عنصر `<textarea>` لحفظ محتوى الملف الذي يُعدَّل، ويمكن جلب محتوى الملف الحالي باستخدام `GET` الذي يستخدِم `fetch`، كما تستطيع استخدام الروابط النسبية مثل index.html بدلًا من http://localhost:8000/index.html للإشارة إلى الملفات التي على الخادم نفسه الذي عليه السكربت العاملة، وإذا نقر المستخدِم على زر ما -حيث يمكنك استخدام العنصر `<form>` والحدث `"submit"` لذلك- فأنشئ طلب `PUT` إلى الرابط نفسه بمحتوى `<textarea>` على أساس متن للطلب من أجل حفظ الملف.

يمكنك بعد ذلك إضافة العنصر `<select>` الذي يحتوي على جميع الملفات في المجلد الأعلى للخادم بإضافة عناصر `<option>` التي تحتوي الأسطر المعادة بواسطة طلب `GET` إلى الرابط `/`، وإذا اختار المستخدِم ملفًا آخرًا -أي الحدث `"change"` على ذلك الملف-، فيجب على السكربت جلب ذلك الملف وعرضه، ومن أجل حفظ ملف ما استخدِم اسم الملف المحدد حاليًا.

دورة تطوير التطبيقات
باستخدام لغة JavaScript

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- شهادة معتمدة من أكاديمية حسوب
- وصول مدى الحياة لمحتويات الدورة
- إرشادات من المدربين على مدار الساعة
- تحديثات مستمرة على الدورة مجانًا
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

21. بناء موقع عبـر Node.js

لا أعلم بعد النبوة أفضل من بث العلم.

— عبد الله بن المبارك.

يوجد ما يسمى بفعاليات مشـاركة المهـارات، حيث يتحـدث النـاس في كلمـات مـوجزة غـير رسـمية عمـا يفعلونه لينفعوا غيرهم به، فإذا كانت الفعالية حول مشاركة مهارات الزراعة مثلًا فربما يتحدث أحـدهم عن زراعـة الكرفس، أو إذا كنا في مجموعة برمجية فربمـا تخـبر النـاس عن Node.js، كمـا تسـمى مثـل تلـك الاجتماعـات بمجموعات المستخدِمين إذا كانت تتعلق بالحوسبة والتقنية، وهي طريقة فعالـة لتوسـيع الأفـق ومعرفـة جديـد التطورات، أو التعرف على أشخاص جدد لهم الاهتمامات نفسها، وسيكون هدفنا في هـذا الفصـل الأخـير إعـداد موقع لإدارة الكلمات المقدمة في اجتماع لمشاركة المهارات.

لنتخيل مجموعةً صغيرةً من الناس تجتمع بانتظام في مكتب أحـد أعضـائها للحـديث عن ركـوب الـدراجات ذات العجلة الواحدة مثلًا، وقد انتقل من كان ينظم تلك الاجتماعات إلى مدينة أخرى ولم يشغل أحـد مكانـه، لـذا نريد هنا إنشاء نظام يسمح للمشاركين بطلب الحديث ومناقشة الكلمات بين بعضـهم بعضًـا دون منظِّم مركـزي لهم، كما ستكون بعض الشيفرة التي سنكتبها في هـذا الفصـل موجهـةً لبيئـة Node.js كمـا فعلنـا في الفصـل السـابق، فلن تعمـل مباشـرةً في صـفحة HTML العاديـة، ويمكن تحميـل الشـيفرة الكاملـة للمشـروع من https://eloquentJavaScript.net/code/skillsharing.zip.

21.1 التصميم

سيحتوي هذا المشروع على جزء يعمل على الخادم مكتـوب لبيئـة Node وجـزء للعميـل مكتـوب من أجـل المتصفح، ويخزن الخادم بيانات النظام ويعطيها إلى العميل، كما يخدِّم الملفات الـتي تسـتخدِم النظـام الخـاص

بجانب العميل، حيث يحتفظ الخادم بقائمة من الكلمات المقترحة للاجتماع التالي ويعرض العميل تلك القائمة، ويكون لكل كلمة اسم مقدِّمها وعنوانها وملخصها ومصفوفة من التعليقات المرتبطة بها، كما يسمح العميل للمستخدِمين باقتراح كلمات جديدة -أي إضافتها إلى القائمة- وحذف الكلمات والتعليق أيضًا على الكلمات الموجودة، فكلما نفّذ المستخدِم شيئًا من هؤلاء فسينشئ العميل طلب HTTP ليخبر الخادم بذلك.

يهيَّأ التطبيق ليعرض الكلمات المقترحة وتعليقاتها عرضًا حيًا، وكلما أرسل أحد كلمةً جديدةً في مكان ما أو أضاف تعليقًا فيجب على كل من تكون الصفحة مفتوحة عنده رؤية ذلك الحدث، وهنا محل التحدي إذ لا توجد طريقة يفتح بها الخادم اتصالًا مع عميل ولا توجد طريقة مناسبة لنعرف مَن من العملاء ينظرون الآن إلى الموقع، ويسمى حل تلك المشكلة بالاستطلاع المفتوح long polling وهو أحد بواعث تصميم بيئة Node من البداية.

21.2 الاستطلاع المفتوح

نحتاج إلى اتصال بين العميل والخادم كي يستطيع الخادم إخبار العميل مباشرةً بتغير شيء ما، لكن لا تقبل متصفحات الويب الاتصالات عادةً، كما أنّ موجِّهات الإنترنت routers تحجب عادةً مثل تلك الاتصالات عن العملاء، لذا لن نستطيع جعل الخادم يبدأ ذلك الاتصال، لكن نستطيع تهيئة الأمر كي يفتح العميل الاتصال ويحتفظ به لفترة كي يستطيع الخادم استخدامه من أجل إرسال معلومات عند الحاجة، غير أنّ طلب HTTP يسمح بتدفق معلومات بسيطة مثل إرسال العميل لطلب ما ورد الخادم عليه باستجابة لذلك الطلب وحسب.

أما إذا أردنا أكثر من ذلك فثَم تقنية اسمها WebSockets تدعمها أغلب المتصفحات الحديثة وتسهل فتح الاتصالات من أجل تبادل البيانات عشوائي، غير أنها صعبة قليلًا في الاستفادة منها لحالتنا، والبديل الذي سنستخدِمه في هذا الفصل سيكون تقنيةً أبسط وهي الاستطلاع المفتوح، حيث يطلب العميل من الخادم معلومات جديدة باستمرار باستخدام طلبات HTTP العادية، ويماطل الخادم في الاستجابة لتلك الطلبات إذا لم يكن ثمة شيء جديد لإبلاغه، وطالما أنّ العميل يضمن وجود طلب استطلاع وجس مفتوح دائمًا، فسيستقبل معلومات من الخادم بسرعة بعد توفرها، فإذا كان تطبيق مشاركة المهارات مفتوحًا لدى فاطمة في متصفحها، فسيكون ذلك المتصفح قد أنشأ طلبًا من أجل التحديثات وسيكون منتظرًا استجابةً لذلك الطلب، فإذا أرسلت إيمان كلمةً عن قيادة الدراجة هبوطًا على تل شديد الانحدار، فسيلاحظ الخادم انتظار فاطمة تحديثات، ويرسل استجابةً تحتوي على الكلمة الجديدة إلى طلبها المنتظِر، وسيستلم متصفح فاطمة تلك البيانات ويحدِّث الشاشة ليعرض الكلمة.

المعتاد لمثل تلك الطلبات والاتصالات أنها تنقطع بعد مهلة محددة تسمى timeout إذا لم يكن ثمة نشاط أو رد، ولكي نمنع حدوث ذلك هنا فإنّ تقنيات الاستطلاع المفتوح تعيّن وقتًا أقصى لكل طلب، حيث يستجيب الخادم بعده ولا بد حتى لو لم يكن ثمة شيء يبلِّغه، ثم ينشئ العميل بعد ذلك طلبًا جديدًا، وإعادة التشغيل الدورية تلك تجعل التقنية أكثر ثباتًا لتسمح للعملاء بالعودة للاتصال بعد فشل مؤقت في الشبكة أو مشاكل في الخادم، وإذا كان لدينا خادمًا يستخدِم الاستطلاع المفتوح فقد يكون لديه آلاف الطلبات التي تنتظره، مما يعني أنّ اتصالات TCP مفتوحة، وهنا تأتي ميزة Node، إذ تسهِّل إدارة عدة اتصالات دون إنشاء خيط تحكم منفصل لكل اتصال منها.

21.3 واجهة HTTP

ينبغي النظر أولًا قبل تصميم الخادم أو العميل إلى النقطة التي يتلاقى فيها كل منهما، وهي واجهة HTTP التي يتواصلان من خلالها، حيث سنستخدم JSON على أساس صيغة لطلبنا وعلى أساس متن للاستجابة أيضًا، كما سنستفيد من توابع HTTP وترويساته كما في خادم الملفات من الفصل العشرين، وبما أنّ الواجهة تتمحور حول مسار `/talks`، فستُستخدَم المسارات التي لا تبدأ بـ `/talks` لخدمة الملفات الساكنة، وهي شيفرة HTML وجافاسكربت لنظام جانب العميل، فإذا أرسلنا طلب `GET` إلى `/talks` فسيعيد مستند JSON يشبه ما يلي:

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]
```

تُنشأ الكلمة الجديدة بإنشاء طلب `PUT` إلى رابط مثل `/talks/Unituning`، حيث يكون الجزء الذي بعد الشرطة الثانية هو عنوان الكلمة، ويجب احتواء متن طلب `PUT` على كائن JSON يستخدِم الخاصيتين

presenter وsummary، وبما أنّ عناوين الكلمات تحتوي على مسافات ومحارف قد لا تظهر كما يجب لها في الرابط، فيجب ترميز سلاسل العناوين النصية بدالة encodeURIComponent عند بناء مثل تلك الروابط.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

قد يبدو طلب إنشاء كلمة عن الوقوف بالدراجة كما يلي:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Hasan",
 "summary": "Standing still on a unicycle"}
```

تدعم مثل تلك الروابط طلبات GET لجلب تمثيل JSON لكلمة ما وطلبات DELETE لحذف الكلمة، كما تضاف التعليقات إلى الكلمة باستخدام طلب POST إلى رابط مثل /talks/Unituning/comments مع متن JSON يحتوي على الخاصيتين author وmessage.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
 "message": "Will you talk about raising a cycle?"}
```

قد تحتوي طلبات GET إلى /talks على ترويسات إضافية تخبر الخادم بتأخير الإجابة إذا لم تتوفر معلومات جديدة، وذلك من أجل دعم الاستطلاع المفتوح، كما سنستخدِم زوجًا من الترويسات صُممتا أساسًا من أجل إدارة التخزين المؤقت وهما ETag وIf-None-Match، وقد تدرِج الخوادم ترويسة ETag -التي تشير إلى وسم الكتلة Entity Tag- في الاستجابة، بحيث تكون قيمتها سلسلةً نصيةً تعرِّف الإصدار الحالي للمورِد، وقد تنشئ العملاء طلبًا إضافيًا عندما تطلب هذا المورد مرةً ثانيةً من خلال إدراج ترويسة If-None-Match التي تحمل قيمتها السلسلة نفسها؛ أما إذا لم يتغير المورد، فسيستجيب الخادم برمز الحالة 304 والذي يعني "غير معدَّل not modified"، ليخبر العميل أنّ إصداره المخزَّن لا زال هو الإصدار الحالي؛ أما إذا لم يطابق الوسم، فسيستجيب الاستجابة العادية.

نحتاج إلى مثل ذلك النظام لأننا نريد تمكين العميل من إخبار الخادم بإصدار قائمة الكلمات التي لديه، وألا يستجيب الخادم إلا عند تغير تلك القائمة، لكن ينبغي على الخادم تأخير الإجابة وعدم الإعادة نهائيًا إلا عند توفر

شيء جديد أو مرور مهلة زمنية محددة بدلًا من إعادة 304 مباشرةً، وعليه فمن أجل تمييز طلبات الاستطلاع المفتوح عن الطلبات الشرطية العادية، فإننا نعطيها ترويسةً أخرى هي `Prefer: wait=90` التي تخبر الخادم باستعداد العميل لانتظار الاستجابة مدةً قدرها 90 ثانية، كما سيحتفظ الخادم برقم إصدار version number يحدِّثه في كل مرة تتغير فيها كلمة ما، وسيستخدم ذلك على أساس قيمة لوسم `ETag`، ويمكن للعملاء إنشاء طلبات مثل هذا ليتم إشعارها عند حدوث تغيير في الكلمة:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(time passes)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

لا يقوم البروتوكول في حالتنا هذه بأيّ تحكم في الوصول، إذ يستطيع أيّ أحد تعليق أو تعديل الكلمات أو يحذفها، وليس من الحكمة وضع نظام مثل هذا على الويب دون حماية إضافية.

21.4 الخادم

لنبدأ ببناء جانب الخادم من البرنامج، حيث ستعمل الشيفرة في هذا القسم على Node.js.

21.4.1 التوجيه Routing

سيستخدم خادمنا `createServer` من أجل بدء خادم HTTP، ويجب علينا التفريق في الدالة التي تعالج طلبًا جديدًا بين أنواع الطلبات المختلفة التي ندعمها وفقًا للتابع والمسار، وصحيح أنه يمكن تنفيذ ذلك بسلسلة طويلة من تعليمات `if`، إلا أنّ طريقة التوجيه أفضل، فالموجّه هو مكون يساعد في إرسال طلب إلى الدالة التي تستطيع معالجته، فنستطيع إخباره أنّ طلبات PUT مثلًا التي يطابق مسارها التعبير النمطي `/^\/talks\/([^\/]+)$/` -يشير إلى `/talks/` متبوعًا بعنوان الكلمة- يمكن معالجتها بدالة ما، كما يساعد على استخراج أجزاء مفيدة من المسار -عنوان الكلمة في حالتنا- مغلَّفًا بين أقواس في التعبير النمطي ثم يمررها إلى الدالة المعالجة.

هناك عدة حزم موجهات جيدة على NPM، لكننا سنكتب واحدةً بأنفسنا لتوضيح الفكرة، وتوضِّح الشيفرة التالية `router.js` الذي سنطلبه من وحدة الخادم الخاص بنا عن طريق `require` لاحقًا:

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
    }
    return null;
  }
};
```

تصدِّر الوحدة صنف `Router`، كما يسمح كائن الموجّه بتسجيل معالِجات جديدة باستخدام التابع `add`، ويمكن حل الطلبات باستخدام التابع `resolve` الخاص به، حيث سيعيد هذا التابع استجابةً عند العثور على معالج، وإذا لم يعثر فسيعيد قيمةً غير معرَّفة `null`، ويجرب طريقًا واحدًا في كل مرة بالترتيب الذي عرِّفَت به تلك الطرق إلى أن يعثر على تطابق، كما تُستدعَى الدوال المعالجة بقيمة `context` التي ستكون نسخة الخادم في حالتنا وسلاسل المطابقة لأيّ مجموعة تعرّفها في تعبيرنا النمطي وكائن الطلب، كما يجب فك تشفير روابط السلاسل النصية بما أنّ الرابط الخام قد يحتوي على رموز من تنسيق `%20`.

21.4.2 تخديم الملفات

إذا لم يطابق الطلب أي نوع معرّف في موجهنا فيجب على الخادم تفسير ذلك على أنه طلب لملف في مجلد `public`، ومن الممكن هنا استخدام خادم الملفات المعرَّف في الفصل العشرين لتقديم مثل تلك الملفات، لكننا لا نحتاج ولا نريد دعم طلبات `PUT` أو `DELETE` على الملفات، ونرغب أن يكون لدينا ميزات مثل

دعم التخزين، وعليه فسنستخدم خادم ملفات ساكنة مجرَّبًا من NPM وليكن `ecstatic` مثلًا، رغم أنه ليس الوحيد على NPM ولكنه يعمل جيدًا ومناسب لأغراضنا.

تصدِّر حزمة `ecstatic` دالةً يمكن استدعاؤها مع كائن تهيئة configuration object لإنتاج دالة معالجة طلبات، وسنستخدِم الخيار `root` لنخبر الخادم بالمكان الذي يجب أعليه البحث فيه عن الملفات، كما تقبل الدالة المعالِجة المعاملَين `request` و`response` ويمكن تمريرهما مباشرةً إلى `createServer` لإنشاء خادم لا يقدم لنا إلا الملفات فقط، كما نريد التحقق أولًا من الطلبات التي يجب معالجتها معالجةً خاصةً، لذا نغلفها في دالة أخرى.

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
          return {body: String(error), status: 500};
        }).then(({body,
                  status = 200,
                  headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
```

```
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

نستخدِم هنا طريقةً للاستجابات تشبه خادم الملفات الذي رأيناه في الفصل السابق، إذ تعيد المعالِجات وعودًا تُحل إلى كائنات تصف الاستجابة، وتغلِّف الخادم في كائن يحمل حالته كذلك.

21.4.3 الكلمات على أساس موارد

تخزِّن الكلمات المقترحة في الخاصية `talks` للخادم، وهو كائن تكون أسماء خصائصه عناوين الكلمات، كما ستُكشف على أساس موارد HTTP تحت `/talks/[title]`، لذا نحتاج إلى إضافة معالجات إلى الموجه الخاص بنا تستخدِم التوابع المختلفة التي تستطيع العملاء استخدامها كي تعمل معها، كما يجب على معالج طلبات `GET` التي تطلب كلمة بعينها البحث عن تلك الكلمة، ويستجيب ببيانات JSON لها أو باستجابة خطأ 404.

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (title in server.talks) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

تُحذَف الكلمة بحذفها من الكائن `talks`.

```
router.add("DELETE", talkPath, async (server, title) => {
  if (title in server.talks) {
    delete server.talks[title];
    server.updated();
  }
  return {status: 204};
});
```

يرسل التابع `updated` -الذي سنعرِّفه لاحقًا- إشعارات إلى طلبات الاستطلاع المفتوح المنتظرة بشأن التغيير، ولجلب محتوى متن الطلب فإننا نعرِّف دالةً تدعى `readStream` تقرأ كل المحتوى من بث قابل للقراءة وتعيد وعدًا يُحل إلى سلسلة نصية.

```
function readStream(stream) {
  return new Promise((resolve, reject) => {
    let data = "";
    stream.on("error", reject);
    stream.on("data", chunk => data += chunk.toString());
    stream.on("end", () => resolve(data));
  });
}
```

أحد المعالِجات التي تحتاج إلى قراءة متون الطلبات هو `PUT` المستخدَم في إنشاء كلمات جديدة، ويجب عليه التحقق من إذا كانت البيانات المعطاة لها الخصائص `presenter` و`summary` والتي تكون سلاسل نصية، فقد تكون أيّ بيانات قادمة من خارج النظام غير منطقية، ولا نريد إفساد نموذج بياناتنا الداخلية أو تعطيله إذا أتت طلبات سيئة bad requests، وإذا بدت البيانات صالحةً، فسيخزِّن المعالِج كائنًا يمثِّل الكلمة الجديدة في كائن `talks`، وهذا سيكتب فوق كلمة موجودة سلفًا في العنوان نفسه ويستدعي `updated` مرةً أخرى.

```
router.add("PUT", talkPath,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let talk;
  try { talk = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!talk ||
```

```
        typeof talk.presenter != "string" ||
        typeof talk.summary != "string") {
      return {status: 400, body: "Bad talk data"};
    }
    server.talks[title] = {title,
                           presenter: talk.presenter,
                           summary: talk.summary,
                           comments: []};
    server.updated();
    return {status: 204};
  });
```

تعمل إضافة تعليق إلى كلمة ما بصورة مشابهة، إذ نستخدِم `readStream` لنحصل على محتوى الطلب ونتحقق من البيانات الناتجة ونخزِّنها على هيئة تعليق إذا كانت صالحةً.

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let comment;
  try { comment = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
    return {status: 400, body: "Bad comment data"};
  } else if (title in server.talks) {
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

إذا حاولنا إضافة تعليق إلى كلمة غير موجودة فسنحصل على الخطأ 404.

21.4.4 دعم الاستطلاع المفتوح

يُعَدّ الجزء المتعلق بمعالجة الاستطلاع المفتوح في هذا الخادم أمرًا مثيرًا، فقد يكون الطلب GET الآتي إلى /talks طلبًا عاديًا أو طلب استطلاع مفتوح، وسيكون لدينا أماكن عدة يجب علينا فيها إرسال مصفوفة من الكلمات talks إلى العميل، لذا سنعرِّف تابعًا مساعدًا يبني مثل تلك المصفوفة ويدرِج ترويسة Etag في الاستجابة.

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = [];
  for (let title of Object.keys(this.talks)) {
    talks.push(this.talks[title]);
  }
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`,
              "Cache-Control": "no-store"}
  };
};
```

يجب على المعالج النظر في ترويسات الطلب ليرى إذا كانت الترويستان If-None-Match وPrefer موجودتين أم لا، كما تخزِّن Node الترويسات التي تكون أسماؤها حساسةً لحالة الأحرف بأسماء ذات أحرف صغيرة.

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

إذا لم يُعط أيّ وسم أو كان الوسم المعطى لا يطابق إصدار الخادم الحالي، فسيستجيب المعالج بقائمة من الكلمات، وإذا كان الطلب شرطيًا ولم تتغير الكلمات، فسننظر في الترويسة `Prefer` لنرى إذا كان يجب علينا تأخير الاستجابة أم نستجيب فورًا، كما تخزَّن دوال رد النداء للطلبات المؤجلة في مصفوفة `waiting` الخاصة بالخادم كي يستطيع إشعارها عند حدوث شيء ما، ويضبط التابع `waitForChanges` مؤقتًا على الفور للاستجابة برمز الحالة 304 إذا انتظر الطلب لفترة طويلة.

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
};
```

يزيد تسجيل التغيير بالتابع `updated` قيمة الإصدار التي هي قيمة الخاصية `version` ويوقظ جميع الطلبات المنتظِرة.

```
SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

هكذا تكون شيفرة الخادم قد تمت، فإذا أنشأنا نسخةً من `SkillShareServer` وبدأناها عند المنفَذ 8000، فسيخدم خادم HTTP الناتج الملفات من المجلد الفرعي `public` مع واجهة لإدارة الكلمات تحت رابط `/talks`.

```
new SkillShareServer(Object.create(null)).start(8000);
```

21.5 العميل

يتكون جانب العميل من موقع لمشاركة المهارات من ثلاثة ملفات هي صفحة HTML صغيرة وورقة تنسيقات style sheet وملف جافاسكربت.

21.5.1 HTML

يُعَدّ تقديم ملف اسمه `index.html` إحدى الطرق المستخدَمة بكثرة في خوادم الويب عند إنشاء طلب مباشرة إلى مسار موافق لمجلد ما، وتدعم وحدة خادم الملفات التي نستخدمها `exstatic` تلك الطريقة، فإذا أنشئ طلب إلى المسار `/` فسيبحث الخادم عن الملف `./public/index.html` حيث يكون `./public` الجذر الذي أعطيناه إليه، ثم يعيد ذلك الملف إذا وجده، وعلى ذلك فإذا أردنا لصفحة أن تظهر عندما يوجَّه متصفح ما إلى خادمنا، فيجب علينا وضعها في `public/index.html`، حيث يكون ملف index الخاص بنا كما يلي:

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

<script src="skillsharing_client.js"></script>
```

يعرِّف هذا الملف عنوان المستند، ويتضمن ورقة تنسيقات تعرِّف بعض التنسيقات لضمان وجود مسافة بين الكلمات، إضافة إلى أمور أخرى، كما يضيف في النهاية عنوانًا في قمة الصفحة ويحمِّل السكربت التي تحتوي على تطبيق جانب العميل.

21.5.2 الإجراءات

تتكون حالة التطبيق من قائمة من الكلمات واسم المستخدم، كما سنخزِّن ذلك في الكائن `{talks,user}`، ولا نريد السماح لواجهة المستخدِم بتعديل الحالة أو إرسال طلبات HTTP، بل قد تطلق إجراءات تصف ما الذي يحاول المستخدِم فعله،

في حين تأخذ دالة `handleAction` مثل هذا الإجراء وتجعله يحدُث، كما تعالَج تغيرات الحالة في الدالة نفسها بما أنّ تحديثات حالتنا بسيطة جدًا.

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
    return Object.assign({}, state, {user: action.user});
  } else if (action.type == "setTalks") {
    return Object.assign({}, state, {talks: action.talks});
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

سنخزن اسم المستخدِم في `localStorage` كي يمكن استعادتها عند تحميل الصفحة؛ أما الإجراءات التي تحتاج إلى إنشاء الخادم طلبات شبكية باستخدام `fetch` إلى واجهة HTTP التي وصفناها من قبل فسنستخدِم دالةً مغلِّفةً هي `fetchOk` تتأكد من أنّ الوعد المعاد مرفوض إذا أعاد الخادم رمز خطأ.

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

تُستخدَم الدالة المساعدة التالية لبناء رابط لكلمة لها عنوان محدَّد.

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

إذا فشل الطلب فلا نريد أن تظل صفحتنا ساكنةً لا تفعل شيء دون تفسير، لذا نعرِّف دالةً تدعى `reportError` تعرض للمستخدِم صندوقًا حواريًا يخبره أنّ شيئًا خاطئًا قد حدث.

```
function reportError(error) {
  alert(String(error));
}
```

21.5.3 إخراج المكونات Rendering Components

سنستخدِم منظورًا يشبه الذي رأيناه في الفصل التاسع عشر والذي يقسِّم التطبيق إلى مكونات، لكن بما أن بعض تلك المكونات قد لا تحتاج إلى تحديث أبدًا أو تُرسم من جديد في كل مرة تُحدَّث فيها، فسنعرِّف أولئك على أساس دوال تعيد عقدة DOM مباشرةً وليس على أساس أصناف، ويوضِّح المثال التالي مكونًا يعرض حقلًا يمكن للمستخدِم إدخال اسمه فيه.

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

الدالة `elt` المستخدَمة لبناء عناصر DOM هي نفسها التي استخدمناها في الفصل التاسع عشر، وتُستخدَم دالة شبيهة بها لإخراج الكلمات، حيث تتضمن قائمةً من التعليقات واستمارةً من أجل إضافة تعليق جديد.

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

معالِج الحدث `"submit"` يستدعي `form.reset` لمسح محتوى الاستمارة بعد إنشاء الإجراء `"newcomment"`، وعند إنشاء أجزاء متوسطة التعقيد من DOM، فسيبدو هذا التنسيق من البرمجة فوضويًا، وهناك امتداد جافاسكربت واسع الاستخدام رغم أنه ليس قياسيًا ويسمى JSX، حيث يسمح لنا بكتابة HTML في السكربتات الخاصة بك مباشرةً مما يحسِّن من مظهر الشيفرة، لكن يجب علينا تشغيل برنامج ما قبل تشغيل الشيفرة نفسها ليحوّل شيفرة HTML الوهمية تلك إلى استدعاءات لدوال جافاسكربت مثل تلك التي نستخدمها ها هنا؛ أما التعليقات فستكون أبسط في الإخراج.

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

أخيرًا، تُخرَج الاستمارة التي يستطيع المستخدِم استخدامها في إنشاء الكلمة كما يلي:

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Submit a Talk"),
     elt("label", null, "Title: ", title),
     elt("label", null, "Summary: ", summary),
     elt("button", {type: "submit"}, "Submit"));
}
```

21.5.4 الاستطلاع

نحتاج إلى قائمة الكلمات الحالية إذا أردنا بدء التطبيق، وبما أن التحميل الابتدائي متعلق للغاية بعملية الاستطلاع المفتوح إذ يجب استخدام `ETag` من الحمل عند الاستطلاع، فسنكتب دالةً تظل تستطلع الخادم للمسار `/talks` وتستدعي دالة رد نداء عند توفر مجموعة كلمات جديدة.

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
```

```
          headers: tag && {"If-None-Match": tag,
                           "Prefer": "wait=90"}
        });
      } catch (e) {
        console.log("Request failed: " + e);
        await new Promise(resolve => setTimeout(resolve, 500));
        continue;
      }
      if (response.status == 304) continue;
      tag = response.headers.get("ETag");
      update(await response.json());
    }
  }
```

بما أن هذه الدالة هي دالة `async` فمن السهل تنفيذ تكرار حلقي وانتظار الطلب، وهي تشغِّل حلقةً تكراريةً لا نهائيةً تجلب قائمةً من الكلمات في كل تكرار إما جلبًا عاديًا أو مع تضمين الترويسات التي تجعله طلب استطلاع مفتوح إذا لم يكن هذا هو الطلب الأول، حيث تنتظر الدالة عند فشل الطلب لحظةً ثم تحاول مرةً أخرى وهكذا، فإذا انقطع الاتصال لدينا لوهلة ثم عادة مرةً أخرى فسيستطيع البرنامج أن يتعافى ويتابع التحديث، ويكون الوعد المحلول بواسطة `setTimeout` طريقةً لإجبار دالة `async` على الانتظار.

إذا أعاد الخادم استجابة 304 فهذا يعني انتهاء المهلة الزمنية المحددة لطلب استطلاع مفتوح، لذا يجب أن تبدأ الدالة الطلب التالي، فإذا كانت الاستجابة هي 200 العادية، فسيُقرأ متنها على أنه JSON ويمرَّر إلى رد النداء، كما تخزَّن قيمة الترويسة `ETag` من أجل التكرار التالي.

21.5.5 التطبيق

يربط المكون التالي واجهة المستخدِم كلها بعضها ببعض:

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.syncState(state);
```

```
  }

  syncState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

إذا تغيرت الكلمات فسيُعيد هذا المكون رسمها جميعًا، وهذا أمر بسيط حقًا لكنه مضيعة للوقت وسنعود إليه في التدريبات، إذ نستطيع بدء التطبيق كما يلي:

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

إذا شغلنا الخادم وفتحنا نافذتَي متصفح لـ http://localhost:8000 جنبًا إلى جنب، فسيمكنك رؤية كيف أنّ الإجراءات الذي تحدِثه في إحدى النافذتين تظهر مباشرةً في الأخرى.

21.6 تدريبات

ستتضمن التدريبات التالية تعديل النظام المعرّف في هذا الفصل، ولكي تعمل عليها تأكد من تحميل الشيفرة أولًا من هذا الرابط وتكون قد ثبّتَّ Node لديك من موقعها الرسمي، وكذلك اعتماديات المشروع باستخدام الأمر `npm install`.

21.6.1 الحفاظ على البيانات وتخزينها

يحتفظ خادم مشاركة المهارات ببياناته في الذاكرة، وهذا يعني أنه ستضيع كل الكلمات والتعليقات عند تعطله أو إعادة تشغيله لأيّ سبب كان، لذا وسّع الخادم ليخزِّن بيانات الكلمات في القرص، ويعيد تحميل البيانات تلقائيًا عند إعادة تشغيله، ولا تقلق بشأن الكفاءة وإنما افعل أبسط شيء يؤدي الغرض.

إرشادات الحل

أبسط حل لهذا هو ترميز كائن `talks` كله على أنه JSON وإلقائه في ملف بواسطة `writeFile`، وهناك تابع `update` بالفعل يُستدعى في كل مرة تتغير فيها بيانات الخادم، حيث يمكن توسيعه لكتابة البيانات الجديدة على القرص.

اختر اسم ملف وليكن `./talks,json`، ويمكن للخادم أن يحاول في قراءة هذا الملف باستخدام `readFile` عند بدء عمله، وإذا نجح فيمكن للخادم أن يستخدِم محتويات الملف على أساس تاريخ بدء له؛ لكن احذر، فكائن `talks` بدأ على أساس كائن ليس له نموذج أولي كي يمكن استخدام العامل `in` بصورة موثوقة.

ستعيد `JSON.parse` كائنات عادية يكون نموذجها الأولي هو `Object.prototype`، فإذا استخدمت JSON على أساس صيغة ملفات لك، فيجب عليك نسخ خصائص الكائن المعاد بواسطة `JSON.parse` في كائن جديد ليس له نموذج أولي.

21.6.2 إعادة ضبط حقول التعليقات

تعمل إعادة رسم الكلمات كلها لأنك لا تستطيع عادةً معرفة الفرق بين عقدة DOM وبديلها التوأم، لكن هناك استثناءات لهذا، فإذا بدأت كتابة شيء ما في حقل التعليق لكلمة ما في نافذة متصفح ثم أضفت تعليقًا إلى الكلمة نفسها من متصفح آخر، فسيعاد رسم الحقل في النافذة الأولى ليحذف محتواه وتركيزه focus معًا، وسيكون هذا مزعجًا للغاية إذا كان لدينا نقاشًا بين عدة مستخدِمين من حواسيب مختلفة ومتصفحات عدة يضيفون تعليقات في الوقت نفسه، فهل تستطيع إيجاد طريقة لحل هذه المشكلة؟

إرشادات الحل

إنّ أفضل حل لهذا هو جعل مكونات الكلمات كائنات لها التابع `syncState` كي يمكن تحديثها لتعرض نسخةً معدلةً من الكلمة، وتكون الطريقة الوحيدة التي يمكن بها تغير كلمة ما هي بإضافة تعليقات أكثر، وعليه يكون التابع `syncState` بسيطًا نسبيًا هنا؛ أما الجزء الصعب فهو عند تغير قائمة الكلمات، إذ يجب إصلاح قائمة مكونات DOM الموجودة بكلمات من القائمة الجديدة، مما يعني حذف المكونات التي حُذفت كلماتها وتحديث المكونات التي تغيرت كلماتها.

من المفيد عند تنفيذ ذلك الاحتفاظ بهيكل بيانات يخزن مكونات الكلمات تحت عناوين الكلمات نفسها كي تستطيع معرفة إذا كان مكون ما موجودًا بالنسبة لكلمة معطاة أم لا، ثم تكرِّر حلقيًا على المصفوفة الجديدة للكلمات وتزامن المكون الموجود سلفًا لكل واحدة منها أو تنشئ واحدًا جديدًا، ولحذف المكونات بالنسبة للكلمات المحذوفة فيجب عليك التكرار حلقيًا أيضًا على المكونات وتنظر هل لا زالت الكلمات الموافقة لها موجودةً أم لا.

دورة تطوير التطبيقات
باستخدام لغة JavaScript

احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

أحدث إصدارات أكاديمية حسوب

حسوب
HSOUB.COM