الگوریتم ‌ها و ساختمان داده های javaScript (300 ساعت)

[Document subtitle]

Tik

[Email address]

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

نوشتن توابع

[الگوریتم ‌ها و ساختمان داده های javaScript (300 ساعت) 1](file:///F:\freeCodeCamp.docx#_Toc49112206)

[1. مفاهیم اولیه JavaScript 15](#_Toc49112207)

[آشنایی با JavaScript 15](#_Toc49112208)

[نوشتن توضیح برای کد JavaScript 15](#_Toc49112209)

[تمرین1-1 : 15](#_Toc49112210)

[تعریف متغیر ها در javascript 15](#_Toc49112211)

[تمرین1-2: 16](#_Toc49112212)

[ذخیره سازی مقادیر با عملگر انتساب 16](#_Toc49112213)

[تمرین 1-3: 16](#_Toc49112214)

[مقداردهی اولیه متغیرها با عملگر انتساب 17](#_Toc49112215)

[تمرین 1-4: 17](#_Toc49112216)

[متغیرهای غیرمجاز 17](#_Toc49112217)

[تمرین 1-5: 17](#_Toc49112218)

[حساسیت به متن در متغیرها(case Sensitivity) 17](#_Toc49112219)

[تمرین1-6: 18](#_Toc49112220)

[جمع دو عدد در javaScript 18](#_Toc49112221)

[تمرین1-7: 18](#_Toc49112222)

[تفریق دو عدد در javaScript 19](#_Toc49112223)

[تمرین 1-8: 19](#_Toc49112224)

[ضرب دو عدد در javaScript 19](#_Toc49112225)

[تمرین 1-9: 19](#_Toc49112226)

[تقسیم یک عدد به عدد دیگر در javascript 19](#_Toc49112227)

[تمرین 1-10: 19](#_Toc49112228)

[افزایش یک عدد با JavaScript 20](#_Toc49112229)

[تمرین 1-11: 20](#_Toc49112230)

[کاهش یک عدد با JavaScript 20](#_Toc49112231)

[تمرین 1-12: 20](#_Toc49112232)

[ایجاد اعداد اعشاری با javaScript 21](#_Toc49112233)

[تمرین 1-13: 21](#_Toc49112234)

[ضرب دو عدد اعشاری در javaScript 21](#_Toc49112235)

[تمرین 1-14: 21](#_Toc49112236)

[تقسیم دو عدد اعشاری در javaScript 21](#_Toc49112237)

[تمرین 1-15: 21](#_Toc49112238)

[یافتن باقیمانده در JavaScript 21](#_Toc49112239)

[تمرین1-16: 22](#_Toc49112240)

[ترکیب عملگر مساوی با جمع += 22](#_Toc49112241)

[تمرین 1-17: 22](#_Toc49112242)

[ترکیب عملگر مساوی با منهی -= 23](#_Toc49112243)

[تمرین 1-18: 23](#_Toc49112244)

[ترکیب عملگر مساوی با ضرب \*= 23](#_Toc49112245)

[تمرین1-19: 23](#_Toc49112246)

[ترکیب عملگر مساوی با تقسیم /= 24](#_Toc49112247)

[تمرین 1-20: 24](#_Toc49112248)

[تعریف متغیرهای رشته ای 24](#_Toc49112249)

[تمرین 1-21: 24](#_Toc49112250)

[نقل قول های لیترال در رشته ها 24](#_Toc49112251)

[تمرین1-22: 25](#_Toc49112252)

[محصور کردن رشته ها با Single Quotes(') 25](#_Toc49112253)

[تمرین1-23: 26](#_Toc49112254)

[کاراکترهای فرار در رشته ها 26](#_Toc49112255)

[تمرین 1-24: 27](#_Toc49112256)

[اتصال رشته ها با عملگر + (Plus) 27](#_Toc49112257)

[تمرین 1-25: 28](#_Toc49112258)

[اتصال رشته ها با عملگر \*= 28](#_Toc49112259)

[تمرین 1-26: 28](#_Toc49112260)

[ساخت رشته ها با متغیرها 29](#_Toc49112261)

[تمرین 1-27: 29](#_Toc49112262)

[اضافه کردن متغیرها به رشته ها 29](#_Toc49112263)

[تمرین 1-28: 29](#_Toc49112264)

[یافتن طول یک رشته 30](#_Toc49112265)

[تمرین 1-29: 30](#_Toc49112266)

[استفاده از نماد براکت برای یافتن اولین کاراکتر در رشته 30](#_Toc49112267)

[تمرین1-30: 31](#_Toc49112268)

[تعقیرناپذیری رشته ها 31](#_Toc49112269)

[تمرین 1-31: 31](#_Toc49112270)

[استفاده از نماد براکت برای یافتن چندمین کاراکتر در رشته 32](#_Toc49112271)

[تمرین 1-32: 32](#_Toc49112272)

[استفاده از نماد براکت برای یافتن آخرین کاراکتر در رشته 32](#_Toc49112273)

[تمرین 1-33: 32](#_Toc49112274)

[استفاده از نماد براکت برای یافتن چندمین کاراکتر از آخر در رشته 33](#_Toc49112275)

[تمرین1-34: 33](#_Toc49112276)

[تمرین1-35: Word Blanks 34](#_Toc49112277)

[ذخیره چند مقدار دریک متغیر با استفده از آرایه ها در javaScrpt 35](#_Toc49112278)

[تمرین1-36: 35](#_Toc49112279)

[قرار دادن یک آرایه درون آرایه دیگر 35](#_Toc49112280)

[تمرین1-37: 35](#_Toc49112281)

[دسترسی به داده های آرایه با ایندکس ها 35](#_Toc49112282)

[تمرین1-38: 36](#_Toc49112283)

[تعقیر داده های آرایه با ایندکس ها 36](#_Toc49112284)

[تمرین1-39: 36](#_Toc49112285)

[دسترسی به آرایه های چند بعدی با ایندکس ها 36](#_Toc49112286)

[تمرین1-40: 37](#_Toc49112287)

[دستکاری آرایه ها با push() 37](#_Toc49112288)

[تمرین1-41: 37](#_Toc49112289)

[دستکاری آرایه ها با pop() 37](#_Toc49112290)

[تمرین1-42: 38](#_Toc49112291)

[دستکاری آرایه ها با shift() 38](#_Toc49112292)

[تمرین1-43: 38](#_Toc49112293)

[دستکاری آرایه ها با unshift() 39](#_Toc49112294)

[تمرین1-44: 39](#_Toc49112295)

[تمرین 1-45:لیست خرید 39](#_Toc49112296)

[نوشتن کد با قابلیت استفاده مجدد با استفاده از توابع در JavaScript 40](#_Toc49112297)

[تمرین 1-46: 40](#_Toc49112298)

[ارسال مقادیر به توابع با آرگومان ها 40](#_Toc49112299)

[تمرین 1-47: 41](#_Toc49112300)

[متقیرهای سراسری و توابع 41](#_Toc49112301)

[تمرین 1-48: 41](#_Toc49112302)

[متقیرهای محلی و توابع 41](#_Toc49112303)

[تمرین 1-49: 42](#_Toc49112304)

[مقایسه متقیرهای محلی با متغیر های سراسری در توابع 42](#_Toc49112305)

[تمرین 1-50: 43](#_Toc49112306)

[بازگرداندن یک مقدار از یک تابع با return 43](#_Toc49112307)

[تمرین 1-51: 43](#_Toc49112308)

[بازگشت مقدار Undefined از تابع 43](#_Toc49112309)

[تمرین 1-52: 44](#_Toc49112310)

[تخصیص با مقدار بازگشت داده شده 44](#_Toc49112311)

[تمرین 1-53: 45](#_Toc49112312)

[تمرین 1-54: Stand in Line 45](#_Toc49112313)

[مقادیر بولی(Boolean) 46](#_Toc49112314)

[تمرین 1-55: 46](#_Toc49112315)

[استفاده از منطق شرطی با دستورات if 46](#_Toc49112316)

[تمرین 1-56: 47](#_Toc49112317)

[بررسی عملگر برابری 47](#_Toc49112318)

[تمرین 1-57: 48](#_Toc49112319)

[بررسی عملگر برابری دقیق 48](#_Toc49112320)

[تمرین 1-58: 49](#_Toc49112321)

[با مقایسه مقادیر مختلف تمرین کنید 49](#_Toc49112322)

[تمرین 1-59: 50](#_Toc49112323)

[بررسی عملگر نابرابری 50](#_Toc49112324)

[تمرین 1-60: 51](#_Toc49112325)

[بررسی عملگر نابرابری دقیق 51](#_Toc49112326)

[تمرین 1-61: 51](#_Toc49112327)

[بررسی عملگر بزرگتراز 52](#_Toc49112328)

[تمرین 1-62: 52](#_Toc49112329)

[بررسی عملگر بزرگتر مساوی 53](#_Toc49112330)

[تمرین 1-63: 53](#_Toc49112331)

[بررسی عملگر کمتر‌از 53](#_Toc49112332)

[تمرین 1-64: 54](#_Toc49112333)

[بررسی عملگر کمتر‌مساوی 54](#_Toc49112334)

[تمرین 1-65: 54](#_Toc49112335)

[بررسی And منطقی (&&) 55](#_Toc49112336)

[تمرین 1-67: 55](#_Toc49112337)

[بررسی or منطقی (||) 56](#_Toc49112338)

[تمرین 1-67: 56](#_Toc49112339)

[معرفی دستورات else 57](#_Toc49112340)

[تمرین 1-68: 57](#_Toc49112341)

[معرفی دستورات else if 58](#_Toc49112342)

[تمرین 1-69: 58](#_Toc49112343)

[ترتیب منطقی در دستورات else if 58](#_Toc49112344)

[تمرین 1-70: 59](#_Toc49112345)

[زنجیر کردن دستورات if else 60](#_Toc49112346)

[تمرین 1-71: 60](#_Toc49112347)

[تمرین 1-72: کد گلف 61](#_Toc49112348)

[انتخاب از میان گزینه‌های زیاد با دستورات Switch 61](#_Toc49112349)

[تمرین 1-73: 62](#_Toc49112350)

[افزودن یک گزینه پیش‌فرض در دستورات Switch 62](#_Toc49112351)

[تمرین 1-74: 63](#_Toc49112352)

[گزینه‌های چندگانه در دستورات Switch 63](#_Toc49112353)

[تمرین 1-75: 64](#_Toc49112354)

[جایگزینی زنجیرهای if/else با Switch 64](#_Toc49112355)

[تمرین 1-76: 65](#_Toc49112356)

[بازگرداندن مقادیر بولی از توابع 66](#_Toc49112357)

[تمرین 1-77: 66](#_Toc49112358)

[بازگشت الگوی اولیه برای توابع 67](#_Toc49112359)

[تمرین 1-78: 67](#_Toc49112360)

[تمرین 1-79: شمارش کارت‌ها 68](#_Toc49112361)

[ساخت اشیاء JavaScript 69](#_Toc49112362)

[تمرین 1-80: 69](#_Toc49112363)

[دسترسی به خصوصیات شی با علامت نقطه 70](#_Toc49112364)

[تمرین 1-81: 70](#_Toc49112365)

[دسترسی به ویژگی‌های شی با نماد براکت 70](#_Toc49112366)

[تمرین 1-82: 71](#_Toc49112367)

[دسترسی به خصوصیات شی با متغیرها 71](#_Toc49112368)

[تمرین 1-83: 72](#_Toc49112369)

[به روز رسانی ویژگی های شی 72](#_Toc49112370)

[تمرین 1-84: 73](#_Toc49112371)

[اضافه کردن ویژگی‌های جدید به شی JavaScript 73](#_Toc49112372)

[تمرین 1-85: 74](#_Toc49112373)

[حذف کردن ویژگی ها از یک شی JavaScript 74](#_Toc49112374)

[تمرین 1-86: 75](#_Toc49112375)

[استفاده از اشیاء برای جستجو 75](#_Toc49112376)

[تمرین 1-87: 76](#_Toc49112377)

[تست ویژگی های اشیاء 77](#_Toc49112378)

[تمرین 1-88: 77](#_Toc49112379)

[دستکاری اشیاء پیچیده 78](#_Toc49112380)

[تمرین 1-89: 79](#_Toc49112381)

[دسترسی به اشیاء تودرتو 80](#_Toc49112382)

[تمرین 1-90: 80](#_Toc49112383)

[دسترسی به آرایه‌های تودرتو 81](#_Toc49112384)

[تمرین 1-91: 81](#_Toc49112385)

[تمرین 1-92: مجموعه ضبط 82](#_Toc49112386)

[تکرار با حلقه whileدر JavaScript 84](#_Toc49112387)

[تمرین 1-93: 84](#_Toc49112388)

[تکرار با حلقه forدر JavaScript 85](#_Toc49112389)

[تمرین 1-94: 85](#_Toc49112390)

[تکرار با اعداد فرد با حلقه for 85](#_Toc49112391)

[تمرین 1-95: 86](#_Toc49112392)

[شمارش به عقب با حلقه for 86](#_Toc49112393)

[تمرین 1-96: 86](#_Toc49112394)

[پیمایش میان عناصر آرایه با یک حلقه for 87](#_Toc49112395)

[تمرین 1-97: 87](#_Toc49112396)

[حلقه‌های for تودرتو 87](#_Toc49112397)

[تمرین 1-98: 88](#_Toc49112398)

[تکرار با حلقه Do...While جاوااسکریپت 88](#_Toc49112399)

[تمرین 1-99: 89](#_Toc49112400)

[جایگزینی حلقه ها با استفاده از بازگشتی 90](#_Toc49112401)

[تمرین 1-100: 91](#_Toc49112402)

[تمرین 1-101: جستجوی پروفایل 91](#_Toc49112403)

[تولید توابع تصادفی با javaScript 92](#_Toc49112404)

[تمرین 1-102: 92](#_Toc49112405)

[تولید اعداد تصادفی کامل با javaScript 93](#_Toc49112406)

[تمرین 1-103: 93](#_Toc49112407)

[تولید اعداد تصادفی کامل در یک محدوده خاص 93](#_Toc49112408)

[تمرین 1-104: 93](#_Toc49112409)

[استفاده از تابع parseInt 94](#_Toc49112410)

[تمرین 1-105: 94](#_Toc49112411)

[استفاده از تابع parseInt با یک Radix 94](#_Toc49112412)

[تمرین 1-106: 95](#_Toc49112413)

[استفاده از عملگر شرطی (سه گانه) 95](#_Toc49112414)

[تمرین 1-107: 95](#_Toc49112415)

[استفاده از عملگرهای چندشرطی (سه گانه) 96](#_Toc49112416)

[تمرین 1-108: 97](#_Toc49112417)

[استفاده از بازگشتی برای ایجاد یک شمارش معکوس 97](#_Toc49112418)

[تمرین 1-109: 98](#_Toc49112419)

[استفاده از بازگشتی برای ایجاد طیف وسیعی از اعداد 98](#_Toc49112420)

[تمرین 1-110: 98](#_Toc49112421)

[ES6 -2 99](#_Toc49112422)

[آشنایی با چالش های ES6 99](#_Toc49112423)

[تفاوت بین کلمات کلیدی var و let 100](#_Toc49112424)

[تمرین2-1: کد را به روز کنید تا فقط از کلمه کلیدی let استفاده کند. 101](#_Toc49112425)

[مقایسه دامنه های کلمات کلیدی var و let 101](#_Toc49112426)

[تمرین2-2: 102](#_Toc49112427)

[تعریف متغیر فقط خواندنی با کلمه کلیدی const 103](#_Toc49112428)

[تمرین2-3: 104](#_Toc49112429)

[تعقیرآرایه تعریف شده با const 104](#_Toc49112430)

[تمرین2-4: 105](#_Toc49112431)

[جلوگیری از تعقیر شی 105](#_Toc49112432)

[تمرین2-5: 106](#_Toc49112433)

[استفاده از توابع Arrow برای نوشتن توابع ناشناس کوتاه‌ تر 106](#_Toc49112434)

[تمرین2-6: 107](#_Toc49112435)

[نوشتن توابع Arrow با پارامترها 107](#_Toc49112436)

[تمرین2-7: 108](#_Toc49112437)

[تنظیم پارامتر‌های پیش‌فرض برای تابع خود 108](#_Toc49112438)

[تمرین2-8: 108](#_Toc49112439)

[استفاده از پارامتر Rest با پارامتر های تابع 109](#_Toc49112440)

[تمرین2-9: 109](#_Toc49112441)

[استفاده از عملگرSpread برای ارزیابی آرابه ها در محل 109](#_Toc49112442)

[تمرین2-10: 110](#_Toc49112443)

[استفاده از Destructuring Assignment برای استخراج مقدار از شی 110](#_Toc49112444)

[تمرین2-11: 111](#_Toc49112445)

[استفاده از Destructuring Assignment برای انتساب متغیرها از اشیا 111](#_Toc49112446)

[تمرین2-12: 111](#_Toc49112447)

[استفاده از Destructuring Assignment برای انتساب متغیرها از اشیا تودرتو 112](#_Toc49112448)

[تمرین2-13: 112](#_Toc49112449)

[استفاده از Destructuring Assignment برای انتساب متغیرها از آرایه ها 113](#_Toc49112450)

[تمرین2-14: 113](#_Toc49112451)

[استفاده از Destructuring Assignment برای تنظیم مجدد عناصر با استفاده از پارمترRest 113](#_Toc49112452)

[تمرین2-15: 114](#_Toc49112453)

[استفاده از Destructuring Assignment برای ارسال یک شی به عنوان پارامترهای یک تابع 114](#_Toc49112454)

[تمرین2-16: 114](#_Toc49112455)

[ایجاد رشته ها با استفاده از Template Literal ها 115](#_Toc49112456)

[تمرین2-17: 116](#_Toc49112457)

[نوشتن تعریف های لیترال شیء مختصر با استفاده ازکوتاه کردن خاصیت شی 117](#_Toc49112458)

[تمرین2-18: 117](#_Toc49112459)

[نوشتن توابع اعلانی مختصر را با ES6 117](#_Toc49112460)

[تمرین2-19: 118](#_Toc49112461)

[استفاده از نحو class برای تعریف تابع سازنده 118](#_Toc49112462)

[تمرین2-20: 119](#_Toc49112463)

[استفاده *getter* ها و setter ها برای کنترل دسترسی به یک شی 120](#_Toc49112464)

[تمرین2-21: 121](#_Toc49112465)

[ایجاد یک ماجول script 122](#_Toc49112466)

[تمرین2-22: 122](#_Toc49112467)

[استفاده از export برای به اشتراک گذاری یک بلوک کد 122](#_Toc49112468)

[تمرین2-23: 123](#_Toc49112469)

[استفاده مجدد از کد جاوااسکریپت با استفاده از import 123](#_Toc49112470)

[تمرین2-24: 124](#_Toc49112471)

[استفاده از \* برای import کردن همه چیز از فایل 124](#_Toc49112472)

[تمرین2-25: 124](#_Toc49112473)

[ایجاد یک export fallback با export default 125](#_Toc49112474)

[تمرین2-26: 125](#_Toc49112475)

[Import کردن یک default export 125](#_Toc49112476)

[تمرین2-27: 126](#_Toc49112477)

[ایجاد یک promise جاوااسکریپت 126](#_Toc49112478)

[تمرین2-28: 126](#_Toc49112479)

[تکمیل یک promise با resolve و reject 127](#_Toc49112480)

[handle یک promise ، fulfilled با then 128](#_Toc49112481)

[تمرین2-30: 128](#_Toc49112482)

[handle یک promise ، Rejected با catch 128](#_Toc49112483)

[تمرین2-31: 129](#_Toc49112484)

[آشنایی با چالش های عبارات منظم 130](#_Toc49112485)

[استفاده از متد Test 130](#_Toc49112486)

[تمرین3-1: 130](#_Toc49112487)

[مطابقت داشتن با رشته های لیترال 130](#_Toc49112488)

[تمرین3-2: 131](#_Toc49112489)

[مطابقت تعداد مختلف با یک رشته واقعی 131](#_Toc49112490)

[تمرین3-3: 132](#_Toc49112491)

[نادیده گرفتن Case درحین مطابقت 132](#_Toc49112492)

[تمرین3-4: 132](#_Toc49112493)

[مطابقت Exrract 132](#_Toc49112494)

[تمرین3-5: 133](#_Toc49112495)

[یافتن بیشتر از اولین مطابقت 133](#_Toc49112496)

[تمرین3-6: 133](#_Toc49112497)

[هر چیزی را با period، Wildcard مطابقت دهید 134](#_Toc49112498)

[تمرین3-7: 134](#_Toc49112499)

[مطابقت با کاراکترهای مجزا با امکانات متعدد 134](#_Toc49112500)

[تمرین3-8: 135](#_Toc49112501)

[مطابقت با حروف الفبا 135](#_Toc49112502)

[تمرین3-9: 136](#_Toc49112503)

[مطابقت با اعداد و حروف الفبا 136](#_Toc49112504)

[تمرین3-10: 136](#_Toc49112505)

[مطابقت با کاراکترهای منفرد مشخص نشده 136](#_Toc49112506)

[تمرین3-11: 137](#_Toc49112507)

[مطابقت دادن با کاراکترهایی که یک یا چند بار تکرار می شوند 137](#_Toc49112508)

[تمرین3-12: 137](#_Toc49112509)

[کاراکترهایی را که صفر یا چندبارظاهرمی‌شوند را مطابقت دهید 137](#_Toc49112510)

[تمرین3-13: 138](#_Toc49112511)

[کاراکترها را با تطبیق تنبل پیدا کنید 138](#_Toc49112512)

[تمرین3-14: 139](#_Toc49112513)

[مطابقت الگوهای ابتدای رشته 140](#_Toc49112514)

[تمرین3-16: 140](#_Toc49112515)

[مطابقت الگوهای انتهای رشته 140](#_Toc49112516)

[تمرین3-17: 141](#_Toc49112517)

[همه حروف و اعداد را مطابقت دهید 141](#_Toc49112518)

[تمرین3-18: 141](#_Toc49112519)

[مطابقت همه موارد با حروف و اعداد 142](#_Toc49112520)

[تمرین3-19: 142](#_Toc49112521)

[همه شماره ها را مطابقت دهید 142](#_Toc49112522)

[تمرین3-20: 142](#_Toc49112523)

[همه غیر شماره ها را مطابقت دهید 142](#_Toc49112524)

[تمرین3-21: 143](#_Toc49112525)

[تمرین3-22: نام های کاربری ممکن را محدود کنید 143](#_Toc49112526)

[مطابقت با فضای سفید 143](#_Toc49112527)

[تمرین3-23: 144](#_Toc49112528)

[مطابقت با کاراکتر های غیر فضای سفید 144](#_Toc49112529)

[تمرین3-24: 144](#_Toc49112530)

[تعداد بالا و پایین را برای مطابقت مشخص منید 144](#_Toc49112531)

[تمرین3-25: 145](#_Toc49112532)

[فقط تعداد پایین را برای مطابقت مشخص کنید 145](#_Toc49112533)

[تمرین3-26: 146](#_Toc49112534)

[تعداد دقیق مطابقت ها را مشخص کنید 146](#_Toc49112535)

[تمرین3-27: 146](#_Toc49112536)

[بررسی همه یا هیچ کدام 146](#_Toc49112537)

[تمرین3-28: 147](#_Toc49112538)

[Lookahead مثبت و منفی 147](#_Toc49112539)

[تمرین3-29: 148](#_Toc49112540)

[بررسی ترکیبی گروهی از کاراکتر ها 148](#_Toc49112541)

[تمرین3-30: 148](#_Toc49112542)

[استفاده مجدد از الگوها با استفاده از *capture groups* 149](#_Toc49112543)

[تمرین3-31: 149](#_Toc49112544)

[استفاده از گروه های capture برای جستجو و جایگزینی 149](#_Toc49112545)

[تمرین3-32: 150](#_Toc49112546)

[Whitespace را از ابتدا و انتها حذف کنید 150](#_Toc49112547)

[تمرین3-33: 150](#_Toc49112548)

[4-اشکال‌زدایی 151](#_Toc49112549)

[آشنایی با چالش های اشکال زدایی 151](#_Toc49112550)

# 1. مفاهیم اولیه JavaScript

## آشنایی با JavaScript

JavaScript یک زبان برنامه نویسی سطح بالا است که همه مرورگرهای وب مدرن از آن پشتیبانی

می کنند. همچنین یکی از فن آوری های اصلی وب است ، به همراه HTML و CSS که احتمالاً قبلاً یاد گرفته اید. این بخش مفاهیم اصلی برنامه نویسی جاوا اسکریپت ، که از متغیرها و حساب گرفته تا اشیاء و حلقه ها هستند را پوشش می‌دهد.

## نوشتن توضیح برای کد JavaScript

نظرات خطوط کدی هستند که جاوا اسکریپت عمداً آنها را نادیده می گیرد. Comment ها یک روش عالی برای ایجاد توضیحات برای خود و سایر افرادی که بعدا نیاز دارند که بدانند که آن کد چه کاری انجام می‌دهد است. دو روش برای نوشتن توضیحات در جاوا اسکریپت وجود دارد:

با استفاده از // به جاوا اسکریپت گفته می شود که باقی متن در خط فعلی را نادیده بگیرید:

|  |
| --- |
| // This is an in-line comment. |

می توانید نظر چند خطی را با \*/ شروع کرده و با /\* پایان دهید:

|  |
| --- |
| /\* This is a  multi-line comment \*/ |

**بهترین تمرین**

در هنگام نوشتن کد ، باید به طور مرتب توضیحاتی اضافه کنید تا عملکرد بخش هایی از کد خود را روشن کنید. کامنت گذاری خوب می تواند هم برای دیگران و هم برای خودتان در آینده به ارتباط برقرار کردن با هدف کد، کمک کند.

تمرین1-1 :سعی کنید یکی از هر دو نوع Comment هارا ایجاد کنید.

## تعریف متغیر ها در javascript

در علم کامپیوتر ، داده هر چیزی است که برای کامپیوتر معنی دارد. جاوا اسکریپت هشت نوع داده مختلف را ارائه می دهد که که عبارت هستند از: undefined، null، boolean، string، symbol، bigint، number و object.

به عنوان مثال ، رایانه ها بین اعداد مانند شماره 12 و رشته ها مانند "12" ، "dog" یا "123 cat" که مجموعه ای از کاراکترها هستند ، تمایز فائل می شوند. رایانه ها می توانند عملیات ریاضی را بر روی اعداد انجام دهند ، اما روی رشته ها نمی توانند.

متغیرها به رایانه ها اجازه می دهند داده ها را به روشی پویا ذخیره و دستکاری کنند. آنها این کار را با استفاده از "برچسب" انجام می دهند تا به جای استفاده از خود داده ها، به داده ها اشاره کنند. هر یک از این هشت نوع داده ممکن است در یک متغیر ذخیره شود.

متغیرها مثل متغیرهای x و y هستند که در ریاضیات از آنها استفاده می کنید ، به این معنی که آنها یک اسم ساده برای نمایش داده هایی هستند که می خواهیم به آنها مراجعه کنیم. متغیرهای رایانه با متغیرهای ریاضی فرق دارند زیرا می توانند مقادیر مختلفی را در زمان‌های مختلف ذخیره کنند.

ما با قرار دادن کلید واژه var در مقابل نام متغیر به JavaScript می‌گوییم این یک متغیر است:

|  |
| --- |
| var ourName; |

متغیری به نام OurName ایجاد می شود. در جاوا اسکریپت دستورات را با استفاده از ; به پایان می رسانیم. در نام متغیرها می توان از اعداد ، حروف و $ یا \_ استفاده کرد ، اما نباید دارای فاصله(space) باشند یا با یک شماره شروع شوند.

تمرین1-2: برای ایجاد متغیری به نام myName از کلید واژه var استفاده کنید.

**راهنمایی:** اگر گیر افتادید ، به مثال بالای ما نگاه کنید.

## ذخیره سازی مقادیر با عملگر انتساب

در JavaScript ، شما می توانید مقدار را در یک متغیر با عملگر انتساب ذخیره کنید.

myVariable = 5;

این تکه کد عدد 5 را به myVariable تخصیص می دهد. انتساب همیشه از راست به چپ است. قبل از اینکه مقدار به متغیر در سمت چپ عملگر اختصاص یابد ، همه چیز در سمت راست عملگر = حل می شود.

|  |
| --- |
| myVar = 5;  myNum = myVar; |

این قطعه کد 5 را به myVar اختصاص می دهد و سپس myVar را که دارای مقدار 5 است را به myNum اختصاص می دهد.

تمرین 1-3: **مقدار 7 را به متغیر a اختصاص دهید.**

**محتویات a را به متغیر b اختصاص دهید**

## مقداردهی اولیه متغیرها با عملگر انتساب

معمول است که یک متغیر را در همان خطی که متغیر تعریف شده است ، مقداردهی کنید.

var myVar = 0;

یک متغیر جدید به نام myVar ایجاد کرده و مقدار اولیه آن 0 قرار می دهد.

تمرین 1-4:یک متغیر a با var تعریف کنید و مقدار آن را با مقدار 9 مقداردهی کنید.

## متغیرهای غیرمجاز

وقتی متغیرهای جاوا اسکریپت اعلان می‌شوند ، مقدار اولیه undefined را دارند. اگر شما یک عمل ریاضی را بر روی یک متغیر undefined انجام دهید ، نتیجه شما NaN خواهد بود که مخفف

“Not a Number” به معنی "شماره نیست" است.

اگر یک رشته را با یک متغیر undefined را به هم الصاق کنید ، یک رشته لیترال "undefined" دریافت خواهید کرد.

تمرین 1-5:سه متغیر a ، b و c را با 5 ، 10 و " I am a " به ترتیب مقداردهی کنید تا از این رو undefined نشوند.

## حساسیت به متن در متغیرها(case Sensitivity)

در JavaScript نام همه متغیرها و توابع ‘case sensitive’ هستند. این بدان معنی است بزرگ یا کوچک بودن حروف مهم است.

MYVAR همانند MyVar یا myvar نیست. داشتن چندین متغیر مجزا با یک نام اما پوشش متفاوت(بزرگ یا کوچک بودن حروف) ممکن است. اکیداً توصیه می شود که برای صراحت در کدنویسی ، از این ویژگی زبان استفاده نکنید.

**بهترین تمرین**

نام های متغیر را در JavaScript به صورت camelCase بنویسید. در camelCase ، نام متغیر چند کلمه ای است، اولین حرف کلمه را با حروف کوچک و حرف اول هر کلمه بعدی را با حروف بزرگ

می نویسند.

**مثال ها:**

|  |
| --- |
| var someVariable;  var anotherVariableName;  var thisVariableNameIsSoLong; |

تمرین1-6: اعلان‌ها و مقداردهی2‌های موجود را تغییر دهید در نام آنها از camelCase استفاده کند. هیچ متغیر جدیدی ایجاد نکنید.

|  |
| --- |
| // تعریف متغیر ها  var StUdLyCapVaR;  var properCamelCase;  var TitleCaseOver;  // مقداردهی متغیر ها  STUDLYCAPVAR = 10;  PRoperCAmelCAse = "A String";  tITLEcASEoVER = 9000; |

## جمع دو عدد در javaScript

عدد یک نوع داده در JavaScript است که داده های عددی را نشان می دهد. حال بیایید سعی کنیم با استفاده از JavaScript دو ​​عدد را با هم جمع کنیم.

JavaScript از علامت + به عنوان عملگر جمع استفاده می کند که بین دو عدد جای می گیرد.

مثال:

|  |
| --- |
| myVar = 5 + 10; // assigned 15 |

تمرین1-7: 0 را تغییر دهید تا جمع برابر با 20 شود.

|  |
| --- |
| var sum = 10 + 0; |

## تفریق دو عدد در javaScript

همچنین می توانیم یک عدد را از دیگری کم کنیم. جاوا اسکریپت از نماد - برای تفریق استفاده می کند.

مثال:

|  |
| --- |
| myVar = 12 - 6; // assigned 6 |

تمرین 1-8: 0 را طوری تعقیر دهید تا نتیجه 12 شود.

|  |
| --- |
| var difference = 45 - 0; |

## ضرب دو عدد در javaScript

ما همچنین می توانیم یک عدد را به عدد دیگر ضرب کنیم. جاوا اسکریپت برای ضرب دو عدد از نماد \* استفاده می کند.

مثال:

|  |
| --- |
| myVar = 13 \* 13; // assigned 169 |

تمرین 1-9: 0 را طوری تعقیر دهید که حاصل برابر با 80 شود.

|  |
| --- |
| var product = 8 \* 0; |

## تقسیم یک عدد به عدد دیگر در javascript

ما همچنین می توانیم یک عدد را به عدد دیگر تقسیم کنیم. جاوا اسکریپت از نماد / برای تقسیم استفاده

می کند.

مثال:

|  |
| --- |
| myVar = 16 / 2; // assigned 8 |

تمرین 1-10: 0 را طوری تعقیر دهید تا quotient برابر با 2 شود.

|  |
| --- |
| var quotient = 66 / 0; |

## افزایش یک عدد با JavaScript

شما به راحتی می توانید یک متغیر را با عملگر ++ افزایش دهید یا یک واحد اضافه کنید.

i++;

معادل :

i = i + 1;

توجه داشته باشید که کل دستور i++ می شود، و نیاز به علامت مساوی(=) را از بین می برد.

تمرین 1-11: با استفاده از عملگر ++ در کد myVar را تغییر دهید.

|  |
| --- |
| var myVar = 87;  // فقط کد زیر این خط را تعقیر دهید  myVar = myVar + 1; |

## کاهش یک عدد با JavaScript

شما به راحتی می توانید یک متغیر را با عملگر – کاهش دهید یا یک واحد کم کنید.

i--;

معادل:

i = i - 1;

توجه داشته باشید که کل دستور i-- می شود، و نیاز به علامت مساوی(=) را از بین می برد.

تمرین 1-12: با استفاده از عملگر -- در کد myVar را تغییر دهید.

|  |
| --- |
| var myVar = 11;  // فقط کد زیر این خط را تعقیر دهید  myVar = myVar - 1; |

## ایجاد اعداد اعشاری با javaScript

ما می توانیم اعداد اعشاری را نیز در متغیرها ذخیره کنیم. اعداد اعشار گاهی اوقات به عنوان اعداد نقطه شناور یا شناور هم خوانده می شوند.

توجه داشته باشید که همه اعداد واقعی را نمی توان دقیقاً به صورت نقطه شناور نشان داد. این می تواند به خطاهای گرد کردن منجر شود.

تمرین 1-13: یک متغیر myDecimal ایجاد کنید و یک قسمت اعشاری با یک بخش کسری (به عنوان مثال 5.7) به آن بدهید.

|  |
| --- |
| var ourDecimal = 5.7; |

## ضرب دو عدد اعشاری در javaScript

در JavaScript ، دقیقاً مانند اعداد صحیح ، می توانید محاسبات را با اعداد اعشاری انجام دهید. بیایید دو رقم اعشاری را با هم ضرب کنیم تا حاصل آنها بدست بیاید.

تمرین 1-14: 0.0 را طوری تعقیر دهید که حاصل برابر با 5.0 شود.

|  |
| --- |
| var product = 2.0 \* 0.0; |

## تقسیم دو عدد اعشاری در javaScript

حالا بیایید یک عدد اعشار را به عدد دیگری تقسیم کنیم.

تمرین 1-15: 0.0 را طوری تعقیر دهید که quotient برابر با 2.2 شود.

|  |
| --- |
| var quotient = 0.0 / 2.0; // این خط را تعقیر دهید |

## یافتن باقیمانده در JavaScript

عملگر باقی مانده % ، باقی مانده تقسیم دو عدد را بدست می آورد.

مثال:

|  |
| --- |
| 5 % 2 = 1 because Math.floor(5 / 2) = 2 (Quotient) 2 \* 2 = 4 5 - 4 = 1 (Remainder) |

استفاده:

در ریاضیات، با چک کردن باقی مانده تقسیم عدد بر 2 می توان زوج یا فرد بودن عدد را تشخیص داد.

|  |
| --- |
| 17 % 2 = 1 (17 is Odd) 48 % 2 = 0 (48 is Even) |

به یاد داشته باشید که به عملگر باقیمانده (remainder) گاهی اوقات به اشتباه عملگر "modulus" گفته می‌شود. بسیار شبیه به درصد است ، اما با اعداد منفی به درستی کار نمی کند.

تمرین1-16: remainder را برابر با باقی مانده تقسیم 11به 3 با استفاده از عملگر باقیمانده (٪)کنید.

|  |
| --- |
| var remainder; |

## ترکیب عملگر مساوی با جمع +=

در برنامه نویسی ، استفاده از مساوی برای اصلاح محتوای یک متغیر رایج است. به یاد بیاورید که ابتدا همه چیز در سمت راست علامت برابر ارزیابی می شود ، بنابراین می توانیم بگوییم:

myVar = myVar + 5;

5 را به myVar اضافه می کند. از آنجا که این یک الگوی رایج است ، عملگرهایی هستند که هم یک عمل ریاضی و هم تخصیص آن به یک متغیر را در یک مرحله انجام می دهند.

یکی از این عملگرها، عملگر +=است.

|  |
| --- |
| var myVar = 1;  myVar += 5;  console.log(myVar); // Returns 6 |

تمرین 1-17: با استفاده از عملگر += ، مقداردهی های مربوط به a ، b و c را تبدیل کنید.

|  |
| --- |
| var a = 3;  var b = 17;  var c = 12;  a = a + 12;  b = 9 + b;  c = c + 7; |

## ترکیب عملگر مساوی با منهی -=

مانند +=عملگر ، -= یک عدد را از یک متغیر کم می کند.

myVar = myVar - 5;

مقدار 5 رااز myVar کم می کند. این دستور قابل بازنویسی به شکل زیر است:

myVar -= 5;

تمرین 1-18: با استفاده از عملگر -= ، مقداردهی های مربوط به a ، b و c را تبدیل کنید.

|  |
| --- |
| var a = 11;  var b = 9;  var c = 3;  a = a - 6;  b = b - 15;  c = c - 1; |

## ترکیب عملگر مساوی با ضرب \*=

عملگر \*= یک متغیر را با یک عدد ضرب می کند.

myVar = myVar \* 5;

myVar را در 5 ضرب می کند. این دستور قابل بازنویسی به شکل زیر است:

myVar \*= 5;

تمرین1-19: با استفاده از عملگر \*= ، مقداردهی های مربوط به a ، b و c را تبدیل کنید.

|  |
| --- |
| var a = 5;  var b = 12;  var c = 4.6;  a = a \* 5;  b = 3 \* b;  c = c \* 10; |

## ترکیب عملگر مساوی با تقسیم /=

عملگر /= یک متغیر را به عدد دیگری تقسیم می کند.

myVar = myVar / 5;

myVar را با 5 تقسیم می کند. این دستور قابل بازنویسی به شکل زیر است:

myVar /= 5;

تمرین 1-20: با استفاده از عملگر /= ، مقداردهی های مربوط به a ، b و c را تبدیل کنید.

|  |
| --- |
| var a = 48;  var b = 108;  var c = 33;  a = a / 12;  b = b / 4;  c = c / 11; |

## تعریف متغیرهای رشته ای

قبلاً از این کد استفاده کرده ایم

var myName = "your name";

به "your name" اصطلاحا یک لیترال رشته ای گفته می شود. این یک رشته است زیرا مجموعه ای از صفر یا چند کاراکتر است که درون ' ' یا " " محصور شده است.

تمرین 1-21:دو متغیر رشته‌ای جدید ایجاد کنید: myFirstName و myLastName و به ترتیب نام و نام خانوادگی خود رابه آنها اختصاص دهید.

## نقل قول های لیترال در رشته ها

وقتی رشته‌ای را تعریف می کنید باید با " (double quote)یا ' (single quote) آن را شروع و به پایان برسانید. چه اتفاقی می افتد هنگامی که شما نیاز به یک عبارت " یا ' در داخل رشته خود دارید؟

در جاوا اسکریپت ، می توانید با قرار دادن یک backslash (\) در مقابل " ، آن را نمایش دهید.

|  |
| --- |
| var sampleStr = "Alan said, \"Peter is learning JavaScript\"."; |

این به JavaScript علامت می دهد که " فوق، پایان این رشته نیست ، بلکه باید در داخل رشته ظاهر شود. بنابراین اگر می خواهید این رشته را در کنسول چاپ کنید ، چنین چیزی مشاهده خواهید کرد:

|  |
| --- |
| Alan said, "Peter is learning JavaScript". |

تمرین1-22: برای اختصاص رشته ای به متغیر myStr از backslashes استفاده کنید تا چنین چیزی رو کنسول ببینید:

I am a "double quoted" string inside "double quotes".

|  |
| --- |
| var myStr = ""; // این خط را تعقیر دهید |

## محصور کردن رشته ها با Single Quotes(')

مقادیر رشته ها در JavaScript ممکن است با " (double quote)یا ' (single quote) مشخص شوند ، تا زمانی که شما آن ها را با همان نوع شروع و به پایان برساند. برخلاف برخی از زبان های برنامه نویسی دیگر ، نقل قول های تک و جفت در JavaScript یکسان عمل می کنند.

|  |
| --- |
| doubleQuoteStr = "This is a string";  singleQuoteStr = 'This is also a string'; |

دلیل اینکه ممکن است بخواهید از یک نوع کاراکتر نقل قول بیش از دیگری استفاده کنید این است که می‌خواهید از هر دو در رشته استفاده کنید. این ممکن است اتفاق بیفتد اگر می خواهید conversation را به صورت رشته ای ذخیره کنید و conversation را داخل نقل قول ها داشته باشید. استفاده دیگر از آن ذخیره برچسب <a> با ویژگی های مختلف در نقل قول ها ، همه در یک رشته است.

|  |
| --- |
| conversation = 'Finn exclaims to Jake, "Algebraic!"'; |

با این حال ، اگر شما نیاز به استفاده از علامت های نقل قول داخل یکدیگر داشته باشید ، این مسئله مشکل ساز می شود. به یاد داشته باشید ، یک رشته در ابتدا و انتها یک نوع علامت نقل قول را دارد. اما اگر از همان علامت نقل قول را در جایی در وسط رشته خود داشته باشید ، رشته زود متوقف می شود و خطایی به شما نشان میدهد.

|  |
| --- |
| goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';  badStr = 'Finn responds, "Let's go!"'; // Throws an error |

در رشته goodStr فوق ، شما می توانید با استفاده از backslash (\) به عنوان کاراکتر فرار با اطمینان از هر دو علامت نقل قول استفاده کنید.

**توجه داشته باشید**

|  |
| --- |
| Backslash (\) نباید با slash رو به جلو اشتباه گرفته شودّForwardSlash (/) همان کار را انجام نمی دهد. |

تمرین1-23: رشته ارائه شده را با نقل قول های تکی(') در ابتدا و انتها تغییر دهید و هیچ کاراکتر فراری نداشته باشید.

در حال حاضر ، برچسب <a> در رشته از جفت کوتیشن (") در همه جا استفاده می کند. شما باید نقل قول های بیرونی را به تک کوتیشن تغییر دهید تا بتوانید کاراکتر‌های فرار را حذف کنید.

|  |
| --- |
| var myStr = "<a href=\"http://www.example.com\" target=\"\_blank\">Link</a>"; |

## کاراکترهای فرار در رشته ها

کارکترهای نقل قول تنها کاراکترهایی نیستند که می توان درون یک رشته استفاده کرد. دو دلیل برای استفاده از کارکترهای فرار وجود دارد:

1. به شما امکان می دهد از کاراکترهایی استفاده کنید که در حالت عادی قادر به تایپ کردن آنها نیستید ، مانند \r.
2. به شما امکان می دهد چندین علامت نقل قول را در یک رشته استفاده کنیدکه بدون این امکان تفسیر نادرستی از منظور شما می شد.

ما این موضوع را در چالش قبلی آموختیم

|  |  |
| --- | --- |
| کد | خروجی |
| \' | single quote ' |
| \" | double quote " |
| \\ | Backslash \ |
| \n | Newline(خط جدید) |
| \r | carriage return (بازگشت به سر سطر) |
| \t | Tab (فاصله 8 کاراکتری به اندازه یک Tab) |
| \b | word boundary |
| \f | form feed(ابتدای صفحه) |

توجه داشته باشید که برای نمایش یک backslash باید خود آن را بعد از \ قرار داد.

تمرین 1-24: سه خط متن زیر را با استفاده از توالی فرار به متغیر myStr تخصیص دهید.

FirstLine  
    \SecondLine  
ThirdLine

برای درج صحیح کاراکترهای خاص باید از توالی فرار استفاده کنید. همچنین باید از فاصله گذاری همانطور که در بالا به نظر می رسد ، پیروی کنید ، بدون فاصله بین کاراکترهای فرار یا کلمات.

در اینجا متنی است که با کاراکترهای فرار نوشته شده است.

FirstLinenewline tab backslashSecondLinenewlineThirdLine

|  |
| --- |
| var myStr; // این خط را تعقیر دهید |

## اتصال رشته ها با عملگر + (Plus)

در جاوا اسکریپت ، هنگامی که عملگر + با یک مقدار String استفاده می شود ، به آن عملگر الصاق گفته می‌شود. شما می توانید رشته جدیدی را از سایر رشته ها با اتصال آنها با یکدیگر بسازید.

مثال

|  |
| --- |
| 'My name is Alan,' + ' I concatenate.' |

توجه داشته باشید

|  |
| --- |
| مراقب فاصله ها باشید. در هنگام اتصال رشته ها فاصله ها بین رشته های متصل اضافه نمی شوند ، بنابراین باید خودتان آنها را اضافه کنید. |

مثال

|  |
| --- |
| var ourStr = "I come first. " + "I come second.";  // ourStr is "I come first. I come second." |

تمرین 1-25: myStr با استفاده از عملگر + از رشته های "This is the start. " و "This is the end." بسازید.

|  |
| --- |
| var myStr; // فقط این خط را تعقیر دهید |

## اتصال رشته ها با عملگر \*=

ما همچنین می توانیم از عملگر += برای اتصال یک رشته به انتهای یک متغیر رشته ای استفاده کنیم. این کار می تواند برای شکستن یک رشته طولانی در چندین خط بسیار مفید باشد.

مثال

|  |
| --- |
| var ourStr = "I come first. ";  ourStr += "I come second.";  // ourStr is now "I come first. I come second." |

تمرین 1-26: myStr را در چندین خط با اتصال این دو رشته بسازید: "This is the first sentence. " و "This is the second sentence.". با استفاده از عملگر+= شبیه به نحوه نمایش در ویرایشگر استفاده کنید. با اختصاص دادن رشته اول به myStr ، سپس رشته دوم را اضافه کنید.

|  |
| --- |
| var myStr; |

## ساخت رشته ها با متغیرها

گاهی اوقات شما نیاز به ساختن یک رشته به سبک Mad Libs دارید. با استفاده از عملگر جمع (+) ، می توانید یک یا چند متغیر را در رشته ای که در حال ساخت آن هستید وارد کنید.

مثال1:

|  |
| --- |
| var ourName = "freeCodeCamp";  var ourStr = "Hello, our name is " + ourName + ", how are you?";  // ourStr is now "Hello, our name is freeCodeCamp, how are you?" |

تمرین 1-27: myName را با رشته ای برابر با نام خود مقداردهی کنید و myStr را با myName بین رشته های "My name is " و " and I am well!" قرار دهید.

|  |
| --- |
| var myName;  var myStr; |

## اضافه کردن متغیرها به رشته ها

دقیقاً همانطور که می توانیم رشته های مختلفی را از خطوط مختلف بسازیم ، می توانیم متغیرها را با استفاده از عملگر (+=) به یک رشته اضافه کنیم.

مثال:

|  |
| --- |
| var anAdjective = "awesome!";  var ourStr = "freeCodeCamp is ";  ourStr += anAdjective;  // ourStr is now "freeCodeCamp is awesome!" |

تمرین 1-28: someAdjective را مقداردهی کرده و با استفاده از عملگر += آن را به myStr اضافه کنید.

|  |
| --- |
| var someAdjective;  var myStr = "Learning to code is "; |

## یافتن طول یک رشته

شما می توانید طول یک مقدار رشته ای را با نوشتن .length بعد از متغیر رشته ای یا لیترال رشته ای پیدا کنید.

"Alan Peter".length; // 10

به عنوان مثال ، اگر متغیر var firstName = "Charles” را ایجاد کردیم ، می توانیم با استفاده از خاصیت firstName.length بدانیم که رشته "Charles" چقدر طول دارد.

تمرین 1-29: برای مشخص کردن تعداد کاراکترهای متغیر LastName از ویژگی .length استفاده کرده و آن را به lastNameLength اختصاص دهید.

|  |
| --- |
| var lastNameLength = 0;  var lastName = "Lovelace";  // فقط کد زیر این خط را ویرایش کنید  lastNameLength = lastName; |

طول رشته "Lovelace" را یافته و درون متغیر lastNameLength ذخیره می‌کنید.

## استفاده از نماد براکت برای یافتن اولین کاراکتر در رشته

براکت راهی برای بدست آوردن یک کارکتر در یک محل خاص در یک رشته است. زبان های برنامه نویسی مدرن مانند JavaScript ، مثل انسان شمارش را از 1 شروع نمی کنند بلکه آنها از 0 شروع می کنند. به این نمایه سازی(indexing) مبتنی بر صفر گفته می شود.

به عنوان مثال ، کاراکتر در index 0 در کلمه "Charles " ،"C" است. بنابراین اگر "var firstName = "Charles است ، می توانید مقدار حرف اول رشته را با استفاده از [0] firstName پیدا کنیم.

|  |
| --- |
| var firstName = "Charles";  var firstLetter = firstName[0]; // firstLetter is "C" |

مثال:

تمرین1-30: از نماد براکت برای یافتن اولین کاراکتردر متغیر lastName استفاده کنید و آن را به firstLetterOfLastName اختصاص دهید.

راهنمایی: اگر گیر کردید به مثال بالا نگاه کنید.

|  |
| --- |
| var firstLetterOfLastName = "";  var lastName = "Lovelace";  // فقط کد زیر این خط را ویرایش کنید  firstLetterOfLastName = lastName; // این خط را تعقیر دهید |

اولین حرف رشته "Lovelace" را یافته و درون متغیر firstLetterOfLastName ذخیره کنید.

## تعقیرناپذیری رشته ها

در جاوا اسکریپت ، مقادیر رشته ها تغییر ناپذیرند، به این معنی که پس از ایجاد آن‌ها نمی توان آن‌ها را تعقیر داد.

به عنوان مثال کد زیر:

|  |
| --- |
| var myStr = "Bob";  myStr[0] = "J"; |

نمی توان مقدار myStr را به "job" تغییر داد، زیرا محتوای myStr قابل تغییر نیست. توجه داشته باشید که این بدان معنا نیست که myStr قابل تغییر نیست ، فقط این که کاراکترهای فردی یک لیترال رشته‌ای قابل تغییر نیستند.

تنها راه تغییر myStr تخصیص دادن یک رشته جدید به آن است ، مانند این:

|  |
| --- |
| var myStr = "Bob";  myStr = "Job"; |

تمرین 1-31: با استفاده از رویکرد نشان داده شده در مثال بالا ، مقداردهی myStr را تصحیح کنید تا مقدار رشته Hello World را در بر بگیرد.

|  |
| --- |
| var myStr = "Jello World";  myStr[0] = "H"; // این خط را تعقیر دهید |

## استفاده از نماد براکت برای یافتن چندمین کاراکتر در رشته

همچنین می توانید از نماد براکت برای بدست آوردن کاراکتر در موقعیت های دیگر در یک رشته استفاده کنید. به یاد داشته باشید که رایانه ها شمارش را از 0 شروع می کنند ، بنابراین کاراکتر اول در واقع کاراکتر صفر است.

مثال1:

|  |
| --- |
| var firstName = "Ada";  var secondLetterOfFirstName = firstName[1]; // secondLetterOfFirstName is "d" |

تمرین 1-32: بیایید سعی کنیم thirdLetterOfLastName را برابر با حرف سوم متغیر lastName با استفاده از نماد براکت تنظیم کنیم.

راهنمایی: اگر گیر کردید به مثال بالا نگاه کنید.

|  |
| --- |
| var lastName = "Lovelace";  var thirdLetterOfLastName = lastName; // این خط را تعقیر دهید |

## استفاده از نماد براکت برای یافتن آخرین کاراکتر در رشته

برای به دست آوردن آخرین حرف یک رشته ، می توانید 1 را از طول رشته کم کنید.

به عنوان مثال ، اگرvar firstName = "Charles" باشد ، می توانید مقدار حرف آخر رشته را با استفاده از firstName [firstName.length – 1] به دست بیاورید.

مثال:

|  |
| --- |
| var firstName = "Charles";  var lastLetter = firstName[firstName.length - 1]; // lastLetter is "s" |

تمرین 1-33: از نماد براکت برای یافتن آخرین کاراکتر در متغیر lastname استفاده کنید.

راهنمایی: اگر گیر کردید به مثال بالا نگاه کنید.

|  |
| --- |
| var lastName = "Lovelace";  var lastLetterOfLastName = lastName; // این خط را تعقیر دهید |

## استفاده از نماد براکت برای یافتن چندمین کاراکتر از آخر در رشته

شما می توانید از همان اصل استفاده شده برای یافتن آخرین کاراکتر در یک رشته، برای یافتن چندمین کاراکتر از آخر در رشته استفاده کنید. به عنوان مثال ، می توانید مقدار حرف سوم از آخر رشته

var firstName = "Charles" را با استفاده از] firstName[firstName.length - 3 پیدا کنید.

مثال:

|  |
| --- |
| var firstName = "Charles";  var thirdToLastLetter = firstName[firstName.length - 3]; // thirdToLastLetter is "l" |

تمرین1-34: از نماد براکت برای یافتن دومین کاراکتر از آخر متغیر lastname استفاده کنید.

راهنمایی: اگر گیر کردید به مثال بالا نگاه کنید.

|  |
| --- |
| var lastName = "Lovelace";  var secondToLastLetterOfLastName = lastName; // این خط را تعقیر دهید |

### تمرین1-35: Word Blanks

اکنون ما از دانش خود در مورد رشته ها برای ساخت یک بازی کلمه ای به سبک "Mad Libs" استفاده می کنیم که آن را "Blanks Word" می نامیم. با پر کردن جاهای خالی جمله ای را به صورت " طنز آمیز" ایجاد خواهید کرد.

در یک بازی سبک "Mad Libs" به شما جملاتی با کلمات از دست رفته مانند اسم ها ، افعال ، صفت ها و قیدها داده می شود. سپس شما قطعات گمشده را با كلمات موردعلاقه خود به روشی پر می كنید كه جمله تکمیل شده با معنا باشند.

این جمله را در نظر بگیرید:

- "It was really \_\_\_\_, and we \_\_\_\_ ourselves \_\_\_\_".

این جمله سه قطعه گمشده دارد - یک صفت ، یک فعل و یک قید ، و ما می توانیم کلمات مورد نظر خود را برای تکمیل به آن اضافه کنیم. سپس می توانیم جمله کامل شده را به صورت زیر به یک متغیر اختصاص دهیم:

|  |
| --- |
| var sentence = "It was really " + "hot" + ", and we " + "laughed" + " ourselves " + "silly" + "."; |

در این چالش ، ما برای شما یک اسم ، یک فعل ، یک صفت و یک قید را ارائه می دهیم. شما باید با استفاده از کلمات مورد نظر خود ، همراه با کلماتی که ما ارائه می دهیم ، یک جمله را کامل کنید.

برای ساختن یک رشته جدید ، با استفاده از متغیرهای ارائه شده ، باید از عملگر اتصال رشته + استفاده کنید: myNoun ، myAdjective ، myVerb و myAdverb.

سپس رشته تکمیل یافته را به متغیر wordBlanks تخصیص خواهید داد. شما نباید کلمات اختصاص داده شده به متغیرها را تغییر دهید.

همچنین شما باید فاصله هایی برای رشته خود در نظر بگیرید تا جمله آخر بین همه کلمات فاصله داشته باشد. نتیجه باید یک جمله کامل باشد.

|  |
| --- |
| var myNoun = "dog";  var myAdjective = "big";  var myVerb = "ran";  var myAdverb = "quickly";  // فقط کد زیر این خط را تعقیر دهید  var wordBlanks = ""; // این خط را تعقیر دهید |

## ذخیره چند مقدار دریک متغیر با استفده از آرایه ها در javaScrpt

با متغیرهای آرایه جاوا اسکریپت ، ما می توانیم چندین قطعه داده را در یک مکان ذخیره کرد.

یک اعلان آرایه را با یک براکت باز ([) شروع می کنند ، آن را با یک براکت بسته شده(]) پایان می دهند و بین هر ورودی کاما می گذارند ، مانند این:

مثال1:

var sandwich = ["peanut butter", "jelly", "bread"].

تمرین1-36: آرایه جدید myArray را طوری تغییر دهید که هم شامل رشته و هم یک عدد (به این ترتیب) باشد.

راهنمایی: اگر گیر کردید به مثال بالا نگاه کنید.

|  |
| --- |
| var myArray = []; |

## قرار دادن یک آرایه درون آرایه دیگر

همچنین می توانید آرایه ها را درون آرایه دیگری مانند زیر قرار دهید:

var myArray = [["Bulls", 23], ["White Sox", 45]]

به این آرایه چند بعدی نیز گفته می شود.

تمرین1-37: آرایه ای تو در تو به نام myArray ایجاد کنید.

|  |
| --- |
| var myArray = []; |

## دسترسی به داده های آرایه با ایندکس ها

ما می توانیم با استفاده از ایندکس ها به داده های داخل آرایه ها دسترسی پیدا کنیم. ایندکس های آرایه در داخل براکت نوشته شده اند که رشته ها از آن استفاده می کنند ، به جز اینکه به جای مشخص کردن یک کاراکتر ، آنها یک عنصر در آرایه را مشخص می کنند. مانند رشته ها ، آرایه ها از ایندکس گذاری مبتنی بر صفر استفاده می کنند ، بنابراین اولین عنصر در یک آرایه دارای ایندکس 0 است.

مثال:

|  |
| --- |
| var array = [50,60,70];  array[0]; // equals 50  var data = array[1];  // equals 60 |

توجه داشته باشید

|  |
| --- |
| هیچ فاصله ای نباید بین نام آرایه و براکت های مربعی وجود داشته باشد، مثل array [0] .  اگرچه جاوا اسکریپت قادر به پردازش صحیح این موضوع است ، اما ممکن است سایر برنامه نویسان که کد شما را می خوانند گیج شوند. |

تمرین1-38: متغیری به نام myData ایجاد کرده و با استفاده از نماد براکت ، آن را برابر با مقدار اول myArray قرار دهید.

|  |
| --- |
| var myArray = [50,60,70]; |

## تعقیر داده های آرایه با ایندکس ها

برخلاف رشته ها ، مقدارهای آرایه ها قابل تغییر هستند و می توان آزادانه آن ها را تعقیر داد.

مثال:

|  |
| --- |
| var ourArray = [50,40,30];  ourArray[0] = 15; // equals [15,40,30] |

تمرین1-39: داده ه ذخیره شده در index، 0 myArray را به مقدار 45 تغییر دهید.

|  |
| --- |
| var myArray = [18,64,99]; |

## دسترسی به آرایه های چند بعدی با ایندکس ها

یک راه برای فکرکردن به یک آرایه چند بعدی ، در نظر گرفتن آن به عنوان آرایه ای از آرایه ها است. هنگامی که از براکت ها برای دسترسی به آرایه خود استفاده می کنید ، اولین مجموعه براکت ها به ورودی های موجود در آرایه بیرونی (سطح اول) اشاره دارد و هر جفت اضافی از براکت ها به سطح بعدی عناصر داخلی اشاره دارد.

مثال:

|  |
| --- |
| var arr = [    [1,2,3],    [4,5,6],    [7,8,9],    [[10,11,12], 13, 14]  ];  arr[3]; // equals [[10,11,12], 13, 14]  arr[3][0]; // equals [10,11,12]  arr[3][0][1]; // equals 11 |

توجه داشته باشید

|  |
| --- |
| بین نام آرایه و براکت های مربعی ، مانند array [0][0] و حتی این آرایه array [0] [0] مجاز نیست هیچ فاصله ای وجود داشته باشد. اگرچه جاوا اسکریپت قادر به پردازش صحیح این موضوع است ، اما ممکن است سایر برنامه نویسان که کد شما را می خوانند گیج شوند. |

تمرین1-40: با استفاده از نماد براکت ، عنصری از myArray را انتخاب کنید که مانند myData برابر با 8 باشد.

|  |
| --- |
| var myArray = [[1,2,3], [4,5,6], [7,8,9], [[10,11,12], 13, 14]]; // فقط کد زیر این خط را تعقیر دهید  var myData = myArray[0][0]; |

## دستکاری آرایه ها با push()

یک روش آسان برای اضافه کردن داده ها به انتهای یک آرایه از طریق تابع push() است . تابع push() یک یا چند پارامتر را می‌گیرد و آنها را به انتهای آرایه اضافه می‌کند.

مثال ها:

|  |
| --- |
| var arr1 = [1,2,3];  arr1.push(4);  // arr1 is now [1,2,3,4]  var arr2 = ["Stimpson", "J", "cat"];  arr2.push(["happy", "joy"]);  // arr2 now equals ["Stimpson", "J", "cat", ["happy", "joy"]] |

تمرین1-41: ["dog", 3] را به انتهای متغیر myArray اضافه کنید.

|  |
| --- |
| var myArray = [["John", 23], ["cat", 2]]; |

## دستکاری آرایه ها با pop()

راه دیگر برای تغییر داده ها در یک آرایه با تابع pop() است. .pop() برای حذف یک مقدار از انتهای یک آرایه استفاده می شود. ما می توانیم با اختصاص دادن آن به یک متغیر ، این مقدار حذف شده را ذخیره کنیم. به عبارت دیگر ، .pop() آخرین عنصر را از یک آرایه حذف کرده و آن عنصر را برمی گرداند.

هر نوع عنصری را می توان از روی یک آرایه - "اعداد ، رشته ها ، حتی آرایه های تو در تو" حذف کرد.

|  |
| --- |
| var threeArr = [1, 4, 6];  var oneDown = threeArr.pop();  console.log(oneDown); // Returns 6  console.log(threeArr); // Returns [1, 4] |

تمرین1-42: از تابع .pop() برای حذف آخرین مورد از myArray استفاده کنید ، و مقدار "حذف شده" را به removeFromMyArray اختصاص دهید.

|  |
| --- |
| var myArray = [["John", 23], ["cat", 2]];  var removedFromMyArray; |

## دستکاری آرایه ها با shift()

Pop() همیشه آخرین عنصر یک آرایه را حذف می کند. اگر می خواهید اولین مورد را حذف کنید ، چه می کنید؟

اینجاست که shift() وارد می شود. درست مانند .pop() کار می کند ، به جز اینکه اولین عنصر را به جای آخرین حذف می کند.

مثال:

|  |
| --- |
| var ourArray = ["Stimpson", "J", ["cat"]];  var removedFromOurArray = ourArray.shift();  // removedFromOurArray now equals "Stimpson" and ourArray now equals ["J", ["cat"]]. |

تمرین1-43: از تابع .shift() برای حذف اولین مورد از myArray استفاده کنید ، و مقدار "حذف شده" را به removeFromMyArray اختصاص دهید.

|  |
| --- |
| var myArray = [["John", 23], ["cat", 2]];  var removedFromMyArray; |

## دستکاری آرایه ها با unshift()

نه تنها می توانید عناصر را از ابتدای یک آرایه حذف کنید ، بلکه می توانید عناصری را به ابتدای آرایه ای وارد کنید ، یعنی عناصر را در مقابل آرایه اضافه کنید.

.unshift() دقیقاً مانند .push() کار می کند ، اما به جای اضافه کردن عنصر در انتهای آرایه ، unshift() عنصری را به ابتدای آرایه اضافه می کند.

تمرین1-44: ["Paul",35] را به ابتدای متغیر myArray اضافه کنید.

|  |
| --- |
| var myArray = [["John", 23], ["dog", 3]];  myArray.shift(); |

مثال:

|  |
| --- |
| v var ourArray = ["Stimpson", "J", "cat"];  ourArray.shift(); // ourArray now equals ["J", "cat"]  ourArray.unshift("Happy");  // ourArray now equals ["Happy", "J", "cat"] |

### تمرین 1-45:لیست خرید

یک لیست خرید را در متغیر myList ایجاد کنید. این لیست باید یک آرایه چند بعدی باشد که شامل چندین زیر آرایه است. عنصر اول در هر زیر مجموعه باید شامل رشته ای با نام آیتم مورد نظر باشد. عنصر دوم باید عددی باشد که مقدار آن را نشان می دهد. برای مثال:

["Chocolate Bar", 15]

باید حداقل 5 زیر آرایه در لیست وجود داشته باشد.

|  |
| --- |
| var myList = []; |

## نوشتن کد با قابلیت استفاده مجدد با استفاده از توابع در JavaScript

در جاوا اسکریپت ، می توانیم کد خود را به قسمت های قابل استفاده مجدد به نام توابع تقسیم کنیم. در اینجا نمونه ای از توابع آورده شده است:

|  |
| --- |
| function functionName() {    console.log("Hello World");  } |

می توانید با استفاده از نام آن در کنار پرانتز ، این تابع را فراخوانی یا استفاده کنید:

هر بار که تابع functionName(); فراخوانی می شود پیغام "Hello World" روی کنسول دستگاه نمایش داده می‌شود. هر باری که تابع فراخوانی می شود همه کد بین دو علامت { } اجرا خواهد شد.

تمرین 1-46: تابعی با نام reusableFunction ایجاد کنید که " Hi World" را بر روی کنسول دستگاه چاپ می‌کند.

تابع را فراخوانی کنید.

## ارسال مقادیر به توابع با آرگومان ها

پارامترها متغیرهایی هستند که به عنوان جایی برای مقادیری که هنگام فراخوانی در یک تابع قرار می‌گیرند عمل می کنند. وقتی یک تابع تعریف می شود ، به طور معمول همراه با یک یا چند پارامتر تعریف می‌شود.

مقادیر واقعی که هنگام فراخوانی به یک تابع وارد ("یا ارسال") می‌شوند ، به عنوان آرگومان شناخت می‌شوند. در اینجا تابعی با دو پارامتر ، param1 و param2 وجود دارد:

|  |
| --- |
| function testFun(param1, param2) {    console.log(param1, param2);  } |

سپس ما می‌توانیم testFun: testFun("Hello", "World"); را فراخوانی کنیم، ما باید دو آرگومان "Hello" و "World" را به تابع ارسال کنیم . در داخل تابع ، param1 برابر "Hello" و param2 برابر "World" خواهد بود. توجه داشته باشید که می توانید با آرگومان های مختلف دوباره testFun را فراخوانی کنید و پارامترها مقدار آرگومان‌های جدید را خواهند گرفت.

تمرین 1-47: تابعی با نام functionWithArgs ایجاد کنید که دو آرگومان را می پذیرد و جمع آنها را به کنسول دستگاه می دهد.

تابع را با دو عدد به عنوان آرگومان فراخوانی کنید.

## متقیرهای سراسری و توابع

در JavaScript ، دامنه به قابلیت مشاهده متغیرها اشاره دارد. متغیرهایی که خارج از بلوک تابع تعریف می‌شوند دارای دامنه سراسری هستند. این بدان معنی است که آنها را می توان در همه جا در کد JavaScript مشاهده کرد(در همه جا قابل دسترسی هستند).

متغیرهایی که بدون کلمه کلیدی var استفاده می‌شوند به صورت خودکار در محدوده سراسری ایجاد می‌شوند. این می‌تواند عواقب ناخواسته دیگری را در جای دیگری در کد شما یا هنگام اجرای دوباره یک تابع ایجاد کند. شما همیشه باید متغیرهای خود را با var اعلان کنید.

تمرین 1-48: با استفاده از var ، یک متغیر جهانی به نام myGlobal را خارج از هر تابعی اعلان کنید. آن را با مقدار 10 مقداردهی کنید.

در داخل تابع fun1 ، 5 را بدون استفاده از کلید واژه var به oopsGlobal اختصاص دهید.

|  |
| --- |
| // رازیر این خط تعریف کنید myGlobal متغیر  function fun1() {  }  function fun2() {    var output = "";    if (typeof myGlobal != "undefined") {      output += "myGlobal: " + myGlobal;    }    if (typeof oopsGlobal != "undefined") {      output += " oopsGlobal: " + oopsGlobal;    }    console.log(output);  } |

## متقیرهای محلی و توابع

متغیرهایی که در یک تابع اعلان می‌شوند و همچنین پارامترهای توابع دارای دامنه محلی هستند. این بدان معنی است که آنها فقط در این محدوده قابل مشاهده هستند(قابل دسترس هستند).

در اینجا یک تابع myTest با یک متغیر محلی به نام loc وجود دارد.

|  |
| --- |
| function myTest() {    var loc = "foo";    console.log(loc);  }  myTest(); // logs "foo"  console.log(loc); // loc is not defined |

loc در خارج از تابع تعریف نشده است.

تمرین 1-49: ویرایشگر دارای دو console.logs است که به شما کمک می کند تا ببینید چه اتفاقی می افتد. کنسول را بررسی کنید تا ببینید کد شما چگونه تغییر می کند. یک متغیر محلی myVar را درون myLocalScope اعلان کنید و تست ها را اجرا کنید.

توجه: کنسول هنوز دارای 'ReferenceError: myVar is not defined'،خواهد بود ، اما این باعث نمی شود که تست ها شکست بخورند.

|  |
| --- |
| function myLocalScope() {    'use strict';    // فقط کد زیر این خط را تعقیر دهید    console.log('inside myLocalScope', myVar);  }  myLocalScope();  // اجرا کنید و کنسول را چک کنید  // myVar is not defined outside of myLocalScope  console.log('outside myLocalScope', myVar); |

## مقایسه متقیرهای محلی با متغیر های سراسری در توابع

داشتن هر دو متغیر محلی و سراسری با یک نام یکسان ممکن است. وقتی این کار را کردید ، متغیر محلی بر متغیر سراسری ارجعیت دارد.

در این مثال:

|  |
| --- |
| var someVar = "Hat";  function myFun() {    var someVar = "Head";    return someVar;  } //returned "Head" |

تابع muFun ، " Head" را برمی‌گرداند زیرا نسخه محلی متغیر موجود است.

تمرین 1-50: یک متغیر محلی را به تابع myOutfit اضافه کنید تا مقدار "outWear" با "sweater" جایگزین شود.

|  |
| --- |
| var outerWear = "T-Shirt";  function myOutfit() {   return outerWear;  }  myOutfit(); |

## بازگرداندن یک مقدار از یک تابع با return

با آرگومان می توانیم مقادیر را به یک تابع ارسال کنیم. برای ارسال مقداری از تابع ، می توانید از دستور return استفاده کنید.

مثال

|  |
| --- |
| function plusThree(num) {    return num + 3;  }  var answer = plusThree(5); // 8 |

plusThree یک آرگومان برای num می گیرد و مقداری برابر با num + 3 را بر می‌گرداند.

تمرین 1-51: یک تابع TimesFive ایجاد کنید که یک آرگومان را بپذیرد ، آن را 5 برابر کند و مقدار جدید را برگرداند. آخرین نمونه را در ویرایشگر مشاهده کنید و مثالی از چگونگی عملکرد تابع TimesFive خود مشاهده کنید.

## بازگشت مقدار Undefined از تابع

یک تابع می‌تواند شامل دستور return باشد اما اجباری نیست. در صورتی که تابع دستور return نداشته باشد ، وقتی آن‌را صدا می کنید ، عملکرد کد داخلی را پردازش می کند اما مقدار برگشتی Undefined است.

مثال

|  |
| --- |
| var sum = 0;  function addSum(num) {    sum = sum + num;  }  addSum(3); // sum will be modified but returned value is   undefined |

addSum تابعی بدون دستور return است. این تابع متغیر sum سراری را تغییر می دهد اما مقدار بازگشتی تابع undefined است.

تمرین 1-52: یک تابع addFive بدون هیچ آرگومانی ایجاد کنید. این تابع 5 را به متغیر sum اضافه می کند ، اما مقدار برگشتی آن undefined است.

|  |
| --- |
| var sum = 0;  function addThree() {    sum = sum + 3;  }  //اینجا  addThree();  addFive(); |

## تخصیص با مقدار بازگشت داده شده

اگر بحث ما درباره‌ی ذخیره مقادیر با عملگر مساوی را به یاد بیاورید، قبل از تخصیص مقدار همه چیز در سمت راست علامت برابر ارزیابی می‌شد. این بدان معنی است که می توانیم مقدار بازگشت داده شده از یک تابع را بگیریم و آن را به یک متغیر اختصاص دهیم.

فرض کنید ما یک مقدار تابع از پیش تعریف شده داریم که دو عدد را به هم اضافه می کند ، بنابراین:

ourSum = sum(5, 12);

تابع sum را فراخوانی می کند ، که مقدار 17 را برمی گرداند و آن را به متغیر OurSum تخصیص می دهد.

تمرین 1-53: تابع processArg را با آرگومان 7 فراخوانی کنید و مقدار بازگشتی آن را به متغیر processed اختصاص دهید.

|  |
| --- |
| var processed = 0;  function processArg(num) {    return (num + 3) / 5;  } |

## تمرین 1-54: Stand in Line

در علوم رایانه یک صف ساختار داده انتزاعی است که در آن عناصر به ترتیب نگه داشته می شوند. عناصر جدید را می‌توان از پشت صف اضافه کرد و عناصر قدیمی از جلو صف خارج می شوند.

یک تابع nextInLine بنویسید که یک آرایه (arr) و یک عدد را به عنوان آرگومان در نظر می گیرد. عدد را به انتهای آرایه اضافه کنید ، سپس اولین عنصر آرایه را بردارید. سپس تابع nextInLine باید عنصری را که حذف شده بود برگرداند.

|  |
| --- |
| function nextInLine(arr, item) {      return item;  }  // Setup  var testArr = [1,2,3,4,5];  // Display code  console.log("Before: " + JSON.stringify(testArr));  console.log(nextInLine(testArr, 6));  console.log("After: " + JSON.stringify(testArr)); |

## 

## مقادیر بولی(Boolean)

نوع دیگری از داده ها Boolean است. بولین ها فقط مینوانند یک از این دو مقدار باشند:true یا false. آنها در اصل سوئیچ های خاموش-روشن هستند ، که صحیح (true)"روشن" است و غلط(false) "خاموش" است. این دو حالت از هم جدا هستند.

توجه داشته باشید

|  |
| --- |
| مقادیر بولی هرگز با علامت‌های نقل قول نوشته نمی شوند. رشته های "true" و "false" بولی نیستند و معنای خاصی در JavaScript ندارند. |

تمرین 1-55: تابع WelcomeToBooleans را طوری تغییر دهید که هنگام کلیک روی دکمه run ، به جای false، true را برگرداند.

|  |
| --- |
| function welcomeToBooleans() {    return false; // این خط را تعقیر دهید  } |

## استفاده از منطق شرطی با دستورات if

از دستورات if برای تصمیم‌گیری در کد استفاده می شود. کلمه کلیدی if به javaScript می گوید در صورتی که شروط تعریف شده در پرانتزها برقرار باشد دستورات داخل بلوک{} را اجرا کن. این شرایط به عنوان شرایط بولی شناخته شده اند و فقط ممکن است true یا false باشند.

هنگامی که شرط true ارزیابی شود ، برنامه دستورات داخل بلوک{} را اجرا می کند. هنگامی که وضعیت بولی false ارزیابی شود ، دستورات داخل بلوک{} اجرا نمی شود.

شبه کد:

|  |
| --- |
| if (condition is true) {   statement is executed } |

مثال

|  |
| --- |
| function test (myCondition) {    if (myCondition) {       return "It was true";    }    return "It was false";  }  test(true);  // returns "It was true"  test(false); // returns "It was false" |

هنگامی که test با مقدار true فراخوانی می شود ، دستور if، myCondition را ارزیابی می کند تا ببیند true است یا نه. از آنجا که این true است ، تابع "It was true" را برمی‌گرداند. وقتی test را با مقدار false فراخوانی می کنیم ، myCondition ، true نیست و دستور موجود در بلوک{} اجرا نمی شود و تابع "It was false" را برمی‌گرداند.

تمرین 1-56: دستور if ای ایجاد کنید که اگر پارامتر wasThatTrue صحیح است "Yes, that was true" را برگرداند و در غیر این صورت "No, that was false"را برگرداند.

|  |
| --- |
| function trueOrFalse(wasThatTrue) {  } |

## بررسی عملگر برابری

تعداد زیادی عملگر مقایسه در JavaScript وجود دارد. همه این عملگر ها یک مقدار true یا false بولی را برمی‌گردانند. اساسی ترین عملگر، عملگر برابری == است. عملگر برابری دو مقدار را با هم مقایسه می کند و اگر برابر باشند مقدار true و در غیر اینصورت مقدار false را بر‌می‌گرداند. توجه داشته باشید که عملگر برابری با مساوی(=) که مقدار در سمت راست عملگر را به متغیری در سمت چپ عملگر اختصاص می‌دهد متفاوت است.

|  |
| --- |
| function equalityTest(myVal) {    if (myVal == 10) {       return "Equal";    }    return "Not Equal";  } |

اگر myVal برابر با 10 باشد ، عملگر برابری true را باز می گرداند ، بنابراین کد موجود در بلوک اجرا می شود ، و تابع "Equal" را برمی‌گرداند. در غیر این صورت ، تابع مقدار " Not Equal" را برمی گرداند. برای اینکه JavaScript بتواند دو نوع داده متفاوت (مثلاً اعداد و رشته ها) را با یکدیگر مقایسه کند ، باید یک نوع را به نوع دیگری تبدیل کند. این با عنوان "تبدیل نوع" شناخته می‌شود. با این حال ، پس از انجام این کار ، می تواند شرایط را به شرح زیر مقایسه کند:

|  |
| --- |
| 1   ==  1   // true  1   ==  2   // false  1   == '1'  // true  "3" ==  3   // true |

تمرین 1-57: عملگر برابری را به خط مشخص شده اضافه کنید تا تابع هنگامی که مقدار معادل 12 باشد ، "Equal" را برگرداند.

|  |
| --- |
| function testEqual(val) {    if (val) { // این خط را تعقیر دهید      return "Equal";    }    return "Not Equal";  }  testEqual(10); // "Not Equal"  testEqual(12); // "Equal"  testEqual("12"); // "Equal" |

## بررسی عملگر برابری دقیق

برابری دقیق (===) مثل عملگر برابری است (==) است. با این حال ، برخلاف عملگر برابری ، که تلاش می کند هر دو مقدار را ابتدا به یک نوع تبدیل کند سپس مقایسه را انجام دهد، عملگر برابری دقیق تبدیل به یک نوع را انجام نمی دهد

.



مثال ها:

|  |
| --- |
| 3 ===  3   // true  3 === '3'  // false |

در مثال دوم ، 3 از نوع Number و "3" از نوع String است.

تمرین 1-58: در دستور if از عملگر برابری دقیق استفاده کنید تا وقتی val دقیقا برابر با 7 شد ، تابع "Equal" برگرداند.

|  |
| --- |
| function testStrict(val) {    if (val) { // این خط را تعقیر دهید      return "Equal";    }    return "Not Equal";  }  testStrict(10); // "Not Equal"  testStrict("7"); // "Not Equal"  testStrict(7); // "Equal" |

## با مقایسه مقادیر مختلف تمرین کنید

در دو چالش گذشته ، در مورد عملگر برابری (==) و عملگر برابری دقیق (===) یاد گرفتیم. بیایید یک بررسی سریع دیگر را با استفاده از این عملگرها انجام دهیم.

اگر مقادیر مقایسه شده از یک نوع نباشند ، عملگر برابری تبدیل نوع را انجام می دهد و سپس مقادیر را ارزیابی می کند. با این حال ، عملگر برابری دقیق ، هر دو نوع داده و مقدار آن را با هم مقایسه می کند ، بدون تبدیل یک نوع به نوع دیگر.

مثال ها:

|  |
| --- |
| 3 == '3'  // returns true because JavaScript performs type conversion from string to number  3 === '3' // returns false because the types are different and type conversion is not performed |

نکته: در javaScript می توانید نوع متغیر یا مقدار را با عملگر typof به شرح زیر تعیین کنید:

|  |
| --- |
| typeof 3   // returns 'number'  typeof '3' // returns 'string' |

تمرین 1-59: تابع compareEquality در ویرایشگر دو مقدار را با استفاده از عملگر برابری مقایسه می کند. تابع را طوری تغییر دهید که فقط"Equal" را زمانی برگرداند که مقادیر کاملاً برابر باشند.

|  |
| --- |
| function compareEquality(a, b) {    if (a == b) { // این خط را تعقیر دهید      return "Equal";    }    return "Not Equal";  }  compareEquality(10, "10"); // "Not Equal"  compareEquality(10, 10); // "Equal" |

## بررسی عملگر نابرابری

عملگر نابرابری (!=) برعکس عملگر برابری(==) است. معنای آن "برابر نیست" است و false را بر می گرداند اگر دو عدد با هم برابر باشند و برعکس زمانی که دو عدد بربر نباشند true را بر می گرداند. مانند عملگر برابری ، عملگر نابرابری نیز انواع داده ها از مقادیر را هنگام مقایسه تبدیل می کند.

مثال ها:

|  |
| --- |
| 1 !=  2     // true  1 != "1"    // false  1 != '1'    // false  1 != true   // false  0 != false  // false |

تمرین 1-60: عملگر نابرابری!= را به دستور if اضافه کنید ، هنگامی که val با 99 برابر نباشد "Not Equal" را برگرداند.

|  |
| --- |
| function testNotEqual(val) {    if (val) { // Change this line      return "Not Equal";    }    return "Equal";  }  testNotEqual(10); |

## بررسی عملگر نابرابری دقیق

عملگر نابرابری دقیق (!==) نقطه منطقی مقابل عملگر برابری دقیق است. این به معنای "کاملاً برابر نیست" است و درصورتی که عملگر برابری دقیق true را برگرداند این عملگر false را بر می گرداند. عملگر نابرابری دقیق تبدیل نوع داده ها را انجام نمی دهد.

مثال ها:

|  |
| --- |
| 3 !==  3   // false  3 !== '3'  // true  4 !==  3   // true |

تمرین 1-61: عملگر نابرابری دقیق!== را به دستور if اضافه کنید ، تا هنگامی که val با 17 دقیقا برابر نباشد تابع "Not Equal" را برگرداند.

|  |
| --- |
| // Setup  function testStrictNotEqual(val) {    if (val) { // Change this line      return "Not Equal";    }    return "Equal";  }  testStrictNotEqual(10); |

## بررسی عملگر بزرگتراز

عملگر بزرگتر از (<) مقادیر دو عدد را مقایسه می کند. اگر عدد سمت چپ بیشتر از عدد در سمت راست باشد ، true را بر می گرداند. در غیر این صورت، false را برمی گرداند. مانند عملگر برابری ، عملگر بزرگتر انواع داده‌های مقادیر را هنگام مقایسه تبدیل می کند.

مثال ها:

|  |
| --- |
| 5   >  3   // true  7   > '3'  // true  2   >  3   // false  '1' >  9   // false |

تمرین 1-62: عملگر بزرگتر را به خطوط نشان داده شده اضافه کنید تا دستورات بازگشت معنی پیدا کنند.

|  |
| --- |
| function testGreaterThan(val) {    if (val) {  // این خط را تعقیر دهید      return "Over 100";    }    if (val) {  // این خط را تعقیر دهید      return "Over 10";    }    return "10 or Under";  }  testGreaterThan(10); |

## بررسی عملگر بزرگتر مساوی

عملگر بزرگتر مساوی (>=) مقادیر دو عدد را با هم مقایسه می کند. اگر عدد سمت چپ از عدد در سمت راست بزرگتر یا مساوی باشد ، true را بر می گرداند. در غیر اینصوررت false را برمی‌گرداند. مانند عملگر برابری ، عملگر بزرگتر‌مساوی انواع داده ها را هنگام مقایسه تبدیل می کند.

مثال ها:

|  |
| --- |
| 6   >=  6   // true  7   >= '3'  // true  2   >=  3   // false  '7' >=  9   // false |

تمرین 1-63: عملگر بزرگترمساوی را به خطوط نشان داده شده اضافه کنید تا دستورات بازگشت معنی پیدا کنند.

|  |
| --- |
| function testGreaterOrEqual(val) {    if (val) {  // این خط را تعقیر دهید      return "20 or Over";    }    if (val) {  // این خط را تعقیر دهید      return "10 or Over";    }    return "Less than 10";  }  testGreaterOrEqual(10); |

## بررسی عملگر کمتر‌از

عملگر کمتر‌از (<) مقادیر دو عدد را با هم مقایسه می کند. اگر عدد سمت چپ کمتر از عدد در سمت راست باشد، true را بر‌می‌گرداند. در غیر اینصورت false را بر‌می‌گرداند. مانند عملگر برابری در هنگام مقایسه انواع داده‌ها را تبدیل می‌کند.

مثال‌ها:

|  |
| --- |
| 2   < 5  // true  '3' < 7  // true  5   < 5  // false  3   < 2  // false  '8' < 4  // false |

تمرین 1-64: عملگر کمتراز را به خطوط نشان داده شده اضافه کنید تا دستورات بازگشت معنی پیدا کنند.

|  |
| --- |
| function testLessThan(val) {    if (val) {  // این خط را تعقیر دهید   return "Under 25";    }    if (val) {  // این خط را تعقیر دهید      return "Under 55";    }    return "55 or Over";  }  testLessThan(10); |

## بررسی عملگر کمتر‌مساوی

عملگر کمتر‌‌مساوی(<=) مقادیر دو عدد را با هم مقایسه می کند. اگر عدد سمت چپ از عدد در سمت راست کمتر یا مساوی باشد ، true را بر می گرداند. در غیر اینصوررت false را برمی‌گرداند. مانند عملگر برابری ، عملگر کمتر‌مساوی انواع داده ها را هنگام مقایسه تبدیل می کند.

مثال‌ها:

|  |
| --- |
| 4   <= 5  // true  '7' <= 7  // true  5   <= 5  // true  3   <= 2  // false  '8' <= 4  // false |

تمرین 1-65: عملگر کمترمساوی را به خطوط نشان داده شده اضافه کنید تا دستورات بازگشت معنی پیدا کنند.

|  |
| --- |
| function testLessOrEqual(val) {    if (val) {  // این خط را تعقیر دهید      return "Smaller Than or Equal to 12";    }    if (val) {  // این خط را تعقیر دهید      return "Smaller Than or Equal to 24";    }    return "More Than 24";  }  testLessOrEqual(10) |

## بررسی And منطقی (&&)

بعضی اوقات شما نیاز دارید بیش از یک چیز را همزمان تست کنید. عملگر And منطقی(&&) در صورتی true برمی‌گرداند که اگر و فقط اگر عملوند‌های چپ و راست هر دو true باشند. همین اثر می تواند به وسیله یک دستور if درون دستور if دیگری دستیابی شود:

|  |
| --- |
| if (num > 5) {    if (num < 10) {      return "Yes";    }  }  return "No"; |

فقط زمانی که num بزرگتر از 5 و کمتر از10 باشد "Yes" را برمی‌گرداند. همین منطق را می‌توان به صورت زیر نوشت:

|  |
| --- |
| if (num > 5 && num < 10) {    return "Yes";  }  return "No"; |

تمرین 1-67: دو دستور if را با یک دستور جایگزین کنید، با استفاده از عملگر && ،که اگر مقدار کمتر از یا مساوی 50 و بیشتر از یا برابر 25 باشد ، "Yes" را بازگرداند. در غیر اینصورت "No" را بازگرداند.

ذ

|  |
| --- |
| function testLogicalAnd(val) {    if (val) {      if (val) {        return "Yes";      }    }    return "No";  }  testLogicalAnd(10); |

## بررسی or منطقی (||)

عملگرor منطقی (||) اگر هر یک از عملوند ها true باشند true را بر‌می‌گرداند درغیراینصورت، false را بر‌می‌گرداند. عملگر or منطقی از دو علامتpipe (||) تشکیل شده است. این کلید به‌طورمعمول میتواند بین کلید های Backspace و Enter کیبورد شما یافت شود.

الگوی زیر باید ازروش‌های قبلی آشنا به نظر برسد:

|  |
| --- |
| testLogicalOr(15);  if (num > 10) {    return "No";  }  if (num < 5) {    return "No";  }  return "Yes"; |

فقط اگر عدد بین5 و10 باشد "Yes" را بر‌‍می‌گرداند. با همین منطق می‌توان به صورت زیر نوشت:

|  |
| --- |
| if (num > 10 || num < 5) {    return "No";  }  return "Yes"; |

تمرین 1-67: دو دستور if را با یک دستور جایگزین کنید که اگر مقدار آن بین 10 تا 20 نباشد ، "Outside"

را بازگرداند. در غیر این صورت ، "Inside" را برگرداند.

|  |
| --- |
| function testLogicalOr(val) {    if (val) {      return "Outside";    }    if (val) {      return "Outside";    }    return "Inside";  } |

## معرفی دستورات else

وقتی شرط یک دستور if صحیح باشد ، بلوک کد موجود در زیر آن اجرا می شود. چه اتفاقی می افتد وقتی که این شرایط نادرست باشد؟ به طور معمول هیچ اتفاقی نمی افتد. با یک دستور else ، یک بلوک کد متناوب قابل اجرا است.

|  |
| --- |
| if (num > 10) {    return "Bigger than 10";  } else {    return "10 or Less";  } |

تمرین 1-68: دستورات if را در یک دستور if /else ترکیب کنید.

|  |
| --- |
| function testElse(val) {    var result = "";    if (val > 5) {      result = "Bigger than 5";    }    if (val <= 5) {      result = "5 or Smaller";    }    return result;  }  testElse(4); |

## معرفی دستورات else if

اگر چندین شرط دارید که باید مورد توجه قرار گیرند ، شما می توانید دستورات if را با دستورات else if به یکدیگر پیوند دهید.

|  |
| --- |
| if (num > 15) {    return "Bigger than 15";  } else if (num < 5) {    return "Smaller than 5";  } else {    return "Between 5 and 15";  } |

تمرین 1-69: منطق را تغییر دهید تا از دستورات else if استفاده کنید

|  |
| --- |
| function testElseIf(val) {    if (val > 10) {      return "Greater than 10";    }    if (val < 5) {      return "Smaller than 5";    }    return "Between 5 and 10";  }  testElseIf(7); |

## ترتیب منطقی در دستورات else if

ترتیب در دستورات if، else if مهم است. این تابع از بالا به پایین اجرا می شود ، احتیاط کنید که کدام دستور اول می‌آید.

این دو تابع را به عنوان مثال در نظر بگیرید. اولی اینجاست:

|  |
| --- |
| function foo(x) {    if (x < 1) {      return "Less than one";    } else if (x < 2) {      return "Less than two";    } else {      return "Greater than or equal to two";    }  } |

و دومی فقط ترتیب دستورات را جابه جا می‌کند:

|  |
| --- |
| function bar(x) {    if (x < 2) {      return "Less than two";    } else if (x < 1) {      return "Less than one";    } else {      return "Greater than or equal to two";    }  } |

با این که این دو تابع تقریباً یکسان به نظر می رسند اگرما یک عدد را به هر دو تابع ارسال کنیم ، خروجی های مختلفی دریافت می کنیم.

|  |
| --- |
| foo(0) // "Less than one"  bar(0) // "Less than two" |

تمرین 1-70: ترتیب منطق را در تابع تغییر دهید تا در همه موارد دستورات صحیح را برگرداند.

|  |
| --- |
| function orderMyLogic(val) {    if (val < 10) {      return "Less than 10";    } else if (val < 5) {      return "Less than 5";    } else {      return "Greater than or equal to 10";    }  }  orderMyLogic(7); |

## زنجیر کردن دستورات if else

دستورات if/else می‌توانند بایک منطق پیچیده به هم زنجیر شوند.در اینجا یک شبه کد از دستورات if/else if چند زنجیره‌ای هست:

|  |
| --- |
| if (condition1) {  statement1  } else if (condition2) {  statement2  } else if (condition3) {  statement3  . . .  } else {  statementN  } |

تمرین 1-71: دستورات if/else if زنجیر وار برای تحقق شروط زیر بنویسید:

num < 5 - return "Tiny"  
num < 10 - return "Small"  
num < 15 - return "Medium"  
num < 20 - return "Large"  
num >= 20 - return "Huge"

|  |
| --- |
| function testSize(num) {    return "Change Me";  }  testSize(7); |

## تمرین 1-72: کد گلف

در بازی گلف ، هر سوراخ یک par دارد که به معنای تعداد استاندارد ضربه هایی است که از یک گلف بازانتظار می رود به توپ بزند(Stroke) تا در آن سوراخ برای تکمیل بازی جای بگیرد. بسته به اینکه تعداد ضربه های شما تا چه حد بالاتر یا پایین تر است ، یک نام مستعار متفاوت وجود دارد.

تابع شما باید par و تعداد ضربه ها(Stroke را به عنوان آرگومان بگیرد. مطابق این جدول رشته های صحیح را برگردانید که نام ضربه ها را به ترتیب اولویت ذکر می کند. بالا (بالاترین) به پایین (پایین ترین):

|  |  |
| --- | --- |
| Strokes | Return |
| 1 | "Hole-in-one!" |
| <= par - 2 | "Eagle" |
| par - 1 | "Birdie" |
| par | "Par" |
| par + 1 | "Bogey" |
| par + 2 | "Double Bogey" |
| >= par + 3 | "Go Home!" |

par و stroke همیشه عددی و مثبت خواهند بود.

|  |
| --- |
| var names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!"];  function golfScore(par, strokes) {    // Only change code below this line    return "Change Me";    // Only change code above this line  }  golfScore(5, 4); |

## انتخاب از میان گزینه‌های زیاد با دستورات Switch

اگر گزینه های زیادی برای انتخاب دارید ، از دستورات Switch استفاده کنید. یک دستور سوئیچ یک مقدار را امتحان می‌کند و می تواند دستورات case زیادی را داشته باشد که مقادیر ممکن مختلفی را تعریف می‌کند. دستورات از اولین case منطبق تا زمان وقوع break اجرا می شوند.

در اینجا نمونه ای از دستور Switch آورده شده است:

|  |
| --- |
| switch(lowercaseLetter) {    case "a":      console.log("A");      break;    case "b":      console.log("B");      break;  } |

مقادیر case با برابری دقیق(===) آزمایش می شوند. Break به javaScript می گوید که اجرا را متوقف کند. اگر break حذف شود دستور بعدی اجرا خواهد شد.

تمرین 1-73: یک دستور Switch بنویسید که val را آزمایش کرده و answer را براساس شروط زیر تنظیم کند:

1 - "alpha"  
2 - "beta"  
3 - "gamma"  
4 - "delta"

|  |
| --- |
| function caseInSwitch(val) {    var answer = "";    // Only change code below this line    // Only change code above this line    return answer;  }  caseInSwitch(1); |

## افزودن یک گزینه پیش‌فرض در دستورات Switch

در دستور Switch ممکن است شما قادر به تعیین همه مقادیر ممکن به عنوان دستورات case نباشید. در عوض، می‌توانید دستور default را اضافه کنید که در صورت عدم یافتن دستورات case منطبق ، اجرا می شود. مانند یک دستور else در آخر یک زنجیره if /else به آن فکر کنید.

یک عبارت پیش فرض باید آخرین مورد باشد.

|  |
| --- |
| switch (num) {  case value1:  statement1;  break;  case value2:  statement2;  break;  ...  default:  defaultStatement;  break;  } |

تمرین 1-74: یک دستور Switch بنویسید که answer را براساس شروط زیر تنظیم کند:

"a" - "apple"  
"b" - "bird"  
"c" - "cat"  
default - "stuff"

|  |
| --- |
| function switchOfStuff(val) {    var answer = "";    // Only change code below this line    // Only change code above this line    return answer;  }  switchOfStuff(1); |

## گزینه‌های چندگانه در دستورات Switch

اگر دستور break از یک case حذف شود ، دستورات case زیر تا زمان وقوع break اجرا می شوند. اگر چندین ورودی با خروجی یکسان دارید ، می توانید آن‌ها را در دستور Switch مانند این نمایش دهید:

|  |
| --- |
| switch(val) {  case 1:  case 2:  case 3:  result = "1, 2, or 3";  break;  case 4:  result = "4 alone";  } |

Case ها‌ی مربوط به 1 ، 2 و 3 همه نتیجه مشابه را به همراه خواهند داشت.

تمرین 1-75: یک دستور Switch بنویسید که answer را براساس شروط زیر تنظیم کند:

1-3 - "Low"  
4-6 - "Mid"  
7-9 - "High"

توجه داشته باشید که برای هر شماره در این دامنه باید case داشته باشید.

|  |
| --- |
| function sequentialSizes(val) {    var answer = "";    // Only change code below this line    // Only change code above this line    return answer;  }  sequentialSizes(1); |

## جایگزینی زنجیرهای if/else با Switch

اگر گزینه های زیادی برای انتخاب داشته باشید، نوشتن یک دستور Switch می‌تواند آسان تر از نوشتن زنجیره‌ای طولانی از دستورات if/else if باشد.

به شرح زیر:

|  |
| --- |
| chainToSwitch(7);  if (val === 1) {    answer = "a";  } else if (val === 2) {    answer = "b";  } else {    answer = "c";  } |

قابل جایگزینی است با:

|  |
| --- |
| switch(val) {    case 1:      answer = "a";      break;    case 2:      answer = "b";      break;    default:      answer = "c";  } |

تمرین 1-76: دستورات if/else if زنجیر شده را به دستورات Switch تبدیل کنید.

|  |
| --- |
| function chainToSwitch(val) {    var answer = "";    // Only change code below this line    if (val === "bob") {      answer = "Marley";    } else if (val === 42) {      answer = "The Answer";    } else if (val === 1) {      answer = "There is no #1";    } else if (val === 99) {      answer = "Missed me by this much!";    } else if (val === 7) {      answer = "Ate Nine";    }    // Only change code above this line    return answer;  }  chainToSwitch(7); |

## بازگرداندن مقادیر بولی از توابع

ممکن است از بررسی با عملگر برابری به یاد بیاورید که همه عملگرهای مقایسه یک مقدار true یا false بولی را برمی گردانند. بعضی اوقات افرد برای انجام مقایسه ، از دستور if/else استفاده می کنند:

|  |
| --- |
| function isEqual(a,b) {    if (a === b) {      return true;    } else {      return false;    }  } |

اما یک راه بهتر برای انجام این کار وجود دارد. از آنجا که === true یا false را بر‌می‌گرداند ، می توانیم نتیجه مقایسه را برگردانیم:

|  |
| --- |
| function isEqual(a,b) {    return a === b;  } |

تمرین 1-77: با حذف دستورات if/else تابع isLess را اصلاح کنید.

|  |
| --- |
| function isLess(a, b) {    // Only change code below this line    if (a < b) {      return true;    } else {      return false;    }    // Only change code above this line  }  isLess(10, 15); |

## بازگشت الگوی اولیه برای توابع

هنگامی که یک دستور return حاصل شد ، اجرای عملکرد فعلی متوقف می شود و کنترل به محل فراخوانی برمی‌گردد.

مثال:

|  |
| --- |
| abTest(2,2);  function myFun() {    console.log("Hello");    return "World";    console.log("byebye")  }  myFun(); |

خروجی های بالا "Hello" به کنسول ، "World" را برمی گرداند ، اما "byebye" هرگز خروجی نمی باشد ، زیرا تابع در دستور return وجود دارد.

تمرین 1-78: تابع را به طور دقیق تغییر دهید تا اگر a یا b کمتر از 0 باشند ، تابع سریعاً با یک مقدار undefined خارج شود.

**راهنما:** به یاد داشته باشید که undefined یک کلمه کلیدی است ، نه یک رشته.

|  |
| --- |
| function abTest(a, b) {    return Math.round(Math.pow(Math.sqrt(a) + Math.sqrt(b), 2);  }  abTest(2,2); |

## تمرین 1-79: شمارش کارت‌ها

در بازی کازینو Blackjack ، یک بازیکن می تواند با پیگیری تعداد نسبی کارتهای بالا و پایین که در میز مانده‌اند، یک امتیاز بالای خانه کسب کند. به این کار شمارش کارت گفته می شود.

داشتن کارت های بالاتر باقیمانده در روی میز، به نفع بازیکن است. به هر کارت مطابق جدول زیر امتیازی اختصاص داده می شود. وقتی count مثبت است ، بازیکن باید زیاد شرط بندی کند. هنگامی که count صفر یا منفی است ، بازیکن باید پایین شرط بندی کند.

|  |  |
| --- | --- |
| Count Change | Cards |
| +1 | 2, 3, 4, 5, 6 |
| 0 | 7, 8, 9 |
| -1 | 10, 'J', 'Q', 'K', 'A' |

شما باید یک تابع برای شمارش کارت را بنویسید. یک پارامتر card دریافت می کند ، که می تواند یک عدد یا یک رشته باشد و متغیر سراسری count را با توجه به ارزش کارت افزایش یا کاهش دهد (جدول را ببینید). در صورت مثبت بودن count ، رشته ای با تعداد فعلی و رشته Bet ، یا اگر صفر باشد یا منفی ، Hold را باز می‌گرداند. شمارش فعلی و تصمیم بازیکن ( bet(شرط بندی) یا hold (نگه داشتن)) باید با یک فضای واحد از هم جدا شود.

مثال خروجی:

-3 Hold  
5 Bet

راهنما:

وقتی مقدار 7 ، 8 یا 9 است count خود را به 0 بازنشانی نکنید.

آرایه ای را برنگردانید.

نقل قول ها (' یا ") در خروجی را شامل نمی شوند.

|  |
| --- |
| var count = 0;  function cc(card) {    return "Change Me";  }  cc(2); cc(3); cc(7); cc('K'); cc('A'); |

## ساخت اشیاء JavaScript

ممکن است قبلاً اصطلاح شی را شنیده باشید. اشیاء شبیه به آرایه ها هستند، به جز اینکه به جای استفاده از ایندکس ها برای دسترسی و تغییر داده های آنها ، از طریق آنچه خواص نامیده می شود به داده ها در اشیاء دسترسی پیدا می کنید.

اشیاء برای ذخیره سازی داده ها به روشی ساختارمند مفید هستند و می توانند اشیاء دنیای واقعی مانند گربه را نشان دهند. در اینجا نمونه ای از شی گربه آورده شده است:

|  |
| --- |
| var cat = {    "name": "Whiskers",    "legs": 4,    "tails": 1,    "enemies": ["Water", "Dogs"]  }; |

در این مثال ، تمام خصوصیات به صورت رشته ها ذخیره می شوند ، مانند - "name" ، "legs" و "tails". با این حال ، شما همچنین می توانید از اعداد به عنوان خاصیت استفاده کنید. حتی می توانید نقل قول های مربوط به خصوصیات رشته ای تک کلمه ای را به شرح زیر حذف کنید:

|  |
| --- |
| var anotherObject = {    make: "Ford",    5: "five",    "model": "focus"  }; |

اگرچه اگر شیء شما دارای خواص غیر رشته ای باشد ، JavaScript به طور خودکار آنها را به صورت رشته ها تایپ می کند.

تمرین 1-80: شیئی را ایجاد کنید که نمایانگر سگی به نام myDog باشد که شامل خواص "name" (یک رشته) ، "legs"، "tails" و "friends" است. شما می توانید این خصوصیات شی را با مقادیردلخواه مورد نظر تنظیم کنید. "name" یک رشته است ، "legs و "tails" عددی هستند ، و "friends" یک آرایه هست.

|  |
| --- |
| var myDog = {}; |

## دسترسی به خصوصیات شی با علامت نقطه

دو راه برای دسترسی به ویژگی های یک شی وجود دارد: علامت نقطه (.) و نماد براکت ([]) ، شبیه به یک آرایه. نماد نقطه چیزی است که وقتی که شما نام خاصیت را می‌دانید و تلاش می‌کنی به آن دسترسی پیدا کنید از آن استفاده می‌کنید.

در اینجا نمونه ای از استفاده از علامت نقطه (.) برای خواندن ویژگی یک شی وجود دارد:

|  |
| --- |
| var myObj = {    prop1: "val1",    prop2: "val2"  };  var prop1val = myObj.prop1; // val1  var prop2val = myObj.prop2; // val2 |

تمرین 1-81: مقادیر خاصیت testObj را با استفاده از علامت نقطه بخوانید. متغیر hatValue را برابر با خاصیت hat شی قرار دهید و متغیر shirtValue را برابر با خاصیت shirt شی قرار دهید.

|  |
| --- |
| var testObj = {    "hat": "ballcap",    "shirt": "jersey",    "shoes": "cleats"  };  var hatValue = testObj;      // این خط را تعقیر دهید  var shirtValue = testObj;    // این خط را تعقیر دهید |

## دسترسی به ویژگی‌های شی با نماد براکت

راه دوم برای دستیابی به خصوصیات یک شیء ، نماد براکت ([]) است. اگر خاصیت شی مورد نظر که برای دسترسی به آن تلاش می کنید ، فاصله ای در نام خود دارد ، باید از نماد براکت استفاده کنید. با این حال ، شما می توانید از نماد براکت در خصوصیات شیء بدون فاصله‌ها هم استفاده کنید.

در اینجا نمونه ای از استفاده نماد براکت برای خواندن ویژگی یک شی وجود دارد:

|  |
| --- |
| var myObj = {    "Space Name": "Kirk",    "More Space": "Spock",    "NoSpace": "USS Enterprise"  };  myObj["Space Name"]; // Kirk  myObj['More Space']; // Spock  myObj["NoSpace"];    // USS Enterprise |

توجه داشته باشید که اسامی ویژگی ها با فاصله های موجود بین آن ها باید در داخل نقل قول (' یا ") باشند.

تمرین 1-82: مقادیر خواص "an entree" و "the drink" از testObj را با استفاده از نماد براکت بخوانید و به ترتیب آنها را به entreeValue و drinkValue اختصاص دهید.:

|  |
| --- |
| var testObj = {    "an entree": "hamburger",    "my side": "veggies",    "the drink": "water"  };  var entreeValue = testObj;   // این خط را تعقیر دهید  var drinkValue = testObj;    // این خط را تعقیر دهید |

## دسترسی به خصوصیات شی با متغیرها

استفاده دیگر از نماد براکت روی اشیاء دسترسی به خاصیتی است که به عنوان مقدار متغیر ذخیره می شود. این می تواند برای تکرار از طریق ویژگی های یک شی یا هنگام دسترسی به جدول جستجو بسیار مفید باشد.

در اینجا مثالی از استفاده از یک متغیر برای دسترسی به یک ویژگی آورده شده است:

|  |
| --- |
| var dogs = {    Fido: "Mutt",  Hunter: "Doberman",  Snoopie: "Beagle"  };  var myDog = "Hunter";  var myBreed = dogs[myDog];  console.log(myBreed); // "Doberman" |

روش دیگری که می توانید از این مفهوم استفاده کنید این است که در طول اجرای برنامه ، اسم خاصیت به صورت پویا جمع آوری شود ، به شرح زیر:

|  |
| --- |
| var someObj = {    propName: "John"  };  function propPrefix(str) {    var s = "prop";    return s + str;  }  var someProp = propPrefix("Name"); // someProp now holds the value 'propName'  console.log(someObj[someProp]); // "John" |

توجه داشته باشید که ما برای دسترسی به خاصیت از نقل قول های اطراف نام متغیر استفاده نمی کنیم زیرا ما از مقدار متغیر استفاده می کنیم نه نام.

تمرین 1-83: از متغیر playerNumber برای جستجوی بازیکن 16 در testObj با استفاده از نماد براکت استفاده کنید. سپس آن نام را به متغیر player اختصاص دهید.

|  |
| --- |
| var testObj = {    12: "Namath",    16: "Montana",    19: "Unitas"  };  var playerNumber;  var player = testObj;  var dogs = {    Fido: "Mutt",  Hunter: "Doberman",  Snoopie: "Beagle"  }; |

## به روز رسانی ویژگی های شی

بعد از ایجاد یک شیء JavaScript ، می توانید ویژگی های آن را در هر زمان درست مانند هر متغیر دیگری به‌روز کنید. برای به روزرسانی می توانید از نماد نقطه یا براکت استفاده کنید.

به عنوان مثال ، بیایید به myDog نگاه کنیم:

|  |
| --- |
| var ourDog = {    "name": "Camper",    "legs": 4,    "tails": 1,    "friends": ["everything!"]}; |

از آنجا که او یک سگ به خصوص خوشحال است ، بیایید نام او را به "Happy Camper" تغییر دهیم. در اینجا نحوه به روزرسانی ویژگی نام شیء آن آورده شده است: ourDog.name = "Happy Camper"; یا.ourDog["name"] = "Happy Camper"; اکنون هنگامی که ما myDog.name ارزیابی می کنیم، به جای اینکه "Camper" بگیریم ، نام جدید او "Happy Camper" را می گیریم.

تمرین 1-84: ویژگی name شیء myDog را به روز کنید. بیایید نام او را از "Coder" به "Happy Coder" تغییر دهیم. می توانید از نماد نقطه یا براکت استفاده کنید.

|  |
| --- |
| var myDog = {    "name": "Coder",    "legs": 4,    "tails": 1,    "friends": ["freeCodeCamp Campers"]  }; |

## اضافه کردن ویژگی‌های جدید به شی JavaScript

می توانید به همان روشی که می توانید اشیا را تعقیر دهید خصوصیات جدیدی را به اشیاء موجود JavaScript اضافه کنید. در اینجا چگونگی اضافه کردن یک ویژگی "bark" به ourDog آورده شده است:

ourDog.bark = "bow-wow";

یا

ourDog["bark"] = "bow-wow";

حالا وقتی myDog.bark را ارزیابی می‌کنیم ، ما به bark آن میرسیم، "bow-wow".

**مثال:**

|  |
| --- |
| var ourDog = {    "name": "Camper",    "legs": 4,    "tails": 1,    "friends": ["everything!"]  };  ourDog.bark = "bow-wow"; |

تمرین 1-85: یک ویژگی "bark" را به myDog اضافه کنید و آن را روی صدای سگ مانند "woof" تنظیم کنید. شما می‌توانید از نماد نقطه یا براکت استفاده کنید**.**

|  |
| --- |
| var myDog = {    "name": "Happy Coder",    "legs": 4,    "tails": 1,    "friends": ["freeCodeCamp Campers"]  }; |

## حذف کردن ویژگی ها از یک شی JavaScript

ما همچنین می توانیم خصوصیاتی را از اشیاء مانند این حذف کنیم:

delete ourDog.bark;

مثال:

|  |
| --- |
| var ourDog = {    "name": "Camper",    "legs": 4,    "tails": 1,    "friends": ["everything!"],    "bark": "bow-wow"  };  delete ourDog.bark; |

پس از آخرین خط نشان داده شده در بالا ، OurDog اینگونه به نظر می رسد:

|  |
| --- |
| {    "name": "Camper",    "legs": 4,    "tails": 1,    "friends": ["everything!"]  } |

تمرین 1-86: ویژگی "tails" را از myDog حذف کنید. شما می توانید از نماد نقطه یا براکت استفاده کنید.

|  |
| --- |
| var myDog = {    "name": "Happy Coder",    "legs": 4,    "tails": 1,    "friends": ["freeCodeCamp Campers"],    "bark": "woof"  }; |

## استفاده از اشیاء برای جستجو

می توان اشیاء را مانند کلید/ مقدار به عنوان یک فرهنگ لغت تصور کرد. اگر داده های جدولی دارید ، می‌توانید از یک شی برای جستجوی مقادیر به جای یک دستور Switch یا یک زنجیره if / else استفاده کنید. این بسیار مفید است وقتی بدانید که داده های ورودی شما محدود به یک محدوده خاص هستند.

در اینجا مثالی از جستجوی ساده الفبای معکوس آورده شده است:

|  |
| --- |
| var alpha = {    1:"Z",    2:"Y",    3:"X",    4:"W",    ...    24:"C",    25:"B",    26:"A"  };  alpha[2]; // "Y"  alpha[24]; // "C"  var value = 2;  alpha[value]; // "Y" |

تمرین 1-87: دستور Switch را به شیء بنام lookup تبدیل کنید. از آن برای جستجوی Val استفاده کنید و رشته مرتبط را به متغیر result اختصاص دهید.

|  |
| --- |
| function phoneticLookup(val) {    var result = "";   switch(val) {      case "alpha":        result = "Adams";        break;      case "bravo":        result = "Boston";        break;      case "charlie":        result = "Chicago";        break;      case "delta":        result = "Denver";        break;      case "echo":        result = "Easy";        break;      case "foxtrot":        result = "Frank";    }    return result;  }  phoneticLookup("charlie"); |

## تست ویژگی های اشیاء

گاهی اوقات مفید است که بررسی کنید که آیا خاصیت یک شیء خاص وجود دارد یا خیر. ما می توانیم از متد .hasOwnProperty(propname) اشیاء استفاده کنیم تا مشخص شود آیا آن شیء دارای آن ویژگی است یا خیر. .hasOwnProperty() بسته به این که آن ویژگی موجود باشد یانه مقدارtrue یا false را برمی‌گرداند.

مثال

|  |
| --- |
| var myObj = {    top: "hat",    bottom: "pants"  };  myObj.hasOwnProperty("top");    // true  myObj.hasOwnProperty("middle"); // false |

تمرین 1-88: تابع checkObj را اصلاح کنید تا تست شود اگریک شیء به تابع ارسال شود (obj) دارای یک خاصیت خاص (checkProp) شود. اگر خاصیت یافت شد ، مقدار آن خاصیت را برگردانید. اگر چنین نیست ، "Not Found" را برگردانید.

|  |
| --- |
| function checkObj(obj, checkProp) {    return "Change Me!";  } |

## دستکاری اشیاء پیچیده

بعضی اوقات ممکن است بخواهید داده ها را در یک ساختار داده انعطاف پذیر ذخیره کنید. یک شیء javaScript یکی از راه های مدیریت داده های انعطاف پذیر است. آنها امکان ترکیب دلخواه از رشته ها ، اعداد ، بولین ها ، آرایه ها ، توابع و اشیاء را فراهم می کنند.

در اینجا نمونه ای از ساختار داده های پیچیده آورده شده است:

|  |
| --- |
| var ourMusic = [    {      "artist": "Daft Punk",      "title": "Homework",      "release\_year": 1997,      "formats": [        "CD",        "Cassette",        "LP"      ],      "gold": true    }  ]; |

این آرایه ای است که درون آن یک شیء وجود دارد. این شیء دارای چندین ابرداده در مورد یک آلبوم است. همچنین دارای آرایه "formats" تو در تو است. اگر می خواهید رکوردهای آلبوم بیشتری اضافه کنید ، می‌توانید این کار را با اضافه کردن رکوردها به آرایه سطح بالا انجام دهید. اشیاء داده ها را در یک ویژگی نگه می دارند ، که دارای یک قالب کلید/ مقدار است. در مثال بالا ، "artist": "Daft Punk" خاصیتی است که دارای کلید "artist"و مقدار "Daft Punk" است. JavaScript Object Notation یا JSON یک فرمت تبادل داده مرتبط است که برای ذخیره داده ها استفاده می ش

|  |
| --- |
| {    "artist": "Daft Punk",    "title": "Homework",    "release\_year": 1997,    "formats": [      "CD",      "Cassette",      "LP"    ],    "gold": true  } |

نکته: شما نیاز به قرار دادن کاما (,) بعد از هر شیء در آرایه دارید ، مگر اینکه آخرین شیء در آرایه باشد.

تمرین 1-89: یک آلبوم جدید به مجموعه myMusic اضافه کنید. و رشته های artist و title ، شماره release\_year و آرایه‌ای از رشته ها به نام formats را اضافه کنید.

|  |
| --- |
| var myMusic = [    {      "artist": "Billy Joel",      "title": "Piano Man",      "release\_year": 1973,      "formats": [        "CD",        "8T",        "LP"      ],      "gold": true    }    // یک رکورد به اینجا اضافه کنید  ]; |

## دسترسی به اشیاء تودرتو

با کنار هم قراردادن نقطه و یا نماد براکت می توان به خصوصیات فرعی اشیاء دست یافت. در اینجا یک شیء تو در تو قرار دارد:

|  |
| --- |
| var gloveBoxContents = undefined; // Change this line  var ourStorage = {    "desk": {      "drawer": "stapler"    },    "cabinet": {      "top drawer": {        "folder1": "a file",        "folder2": "secrets"      },      "bottom drawer": "soda"    }  };  ourStorage.cabinet["top drawer"].folder2;  // "secrets"  ourStorage.desk.drawer; // "stapler" |

تمرین 1-90: به شی myStorage دسترسی پیدا کرده و محتویات ویژگی glove box را به متغیر gloveBoxContents اختصاص دهید. تا آنجایی که ممکن است برای همه خصوصیات از نماد نقطه استفاده کنید ، در غیر این صورت از نماد براکت استفاده کنید.

|  |
| --- |
| // Setup  var myStorage = {    "car": {      "inside": {        "glove box": "maps",        "passenger seat": "crumbs"       },      "outside": {        "trunk": "jack"      }    }  };  var gloveBoxContents = undefined; // این خط را تعقیر دهید |

## دسترسی به آرایه‌های تودرتو

همانطور که در مثال های قبلی دیدیم ، اشیاء می توانند حاوی هر دو اشیاء تو در تو و آرایه های تو در تو باشند. مانند دسترسی به اشیاء تودرتو ، نماد براکت آرایه برای دسترسی به آرایه های تو در تو می تواند استفاده شود.

در اینجا مثالی از چگونگی دستیابی به یک آرایه تو در تو نشان داده شده است:

|  |
| --- |
| var ourPets = [    {      animalType: "cat",      names: [        "Meowzer",        "Fluffy",        "Kit-Cat"      ]    },    {      animalType: "dog",      names: [        "Spot",        "Bowser",        "Frankie"      ]    }  ];  ourPets[0].names[1]; // "Fluffy"  ourPets[1].names[0]; // "Spot" |

تمرین 1-91: درخت دوم را از متغیر myPlants با استفاده از علامت گذاری نقطه و براکت آرایه بازیابی کنید.

|  |
| --- |
| var myPlants = [    {      type: "flowers",      list: [        "rose",        "tulip",        "dandelion"      ]    },    {      type: "trees",      list: [        "fir",        "pine",        "birch"      ]    }  ];  var secondTree = ""; // این خط را تعقیر دهید |

## تمرین 1-92: مجموعه ضبط

به شما یک شی JSON داده می شود که بخشی از مجموعه آلبوم های موسیقی شما را نشان می دهد. هر آلبوم دارای چندین ویژگی و یک شماره شناسه منحصر به فرد به عنوان کلید آن است. همه آلبوم ها اطلاعات کاملی ندارند.

تابعی که شناسه(id) آلبوم (مانند 2548) ، یک ویژگی prop (مانند "artist" یا "tracks") و یک مقدار (مانند "Addicted to Love") را برای تغییر داده های این مجموعه می‌گیرد بنویسید.

اگر prop، "tracks" نیست و value خالی نیست ("") ، value را برای آن ویژگی آلبوم ضبط به‌روز یا تنظیم کنید. تابع شما همیشه باید کل شیء مجموعه را برگرداند.

چندین قانون برای پردازش داده های ناقص وجود دارد:

اگر prop، "tracks" است اما آلبوم خاصیت "tracks" را ندارد ، قبل از افزودن مقدار جدید به خاصیت مربوط به آلبوم ، یک آرایه خالی ایجاد کنید.

اگرprop، "tracks" است و value خالی نیست ("") ، value را به انتهای مجموعه tracks موجود در آلبوم اضافه کنید.

اگر value خالی است ("") ، ویژگی prop داده شده را از آلبوم حذف کنید.

راهنمایی:

هنگام دستیابی به خصوصیات شی با متغیرها ، از نماد براکت استفاده کنید.

متد "push" در اینجا مفید خواهد بود. نحوه عملکرد دستکاری آرایه ها با push() را مجددا بررسی کنید تا ببینید چگونه کار می‌کند.

شما می توانید برای بررسی مجدد به دستکاری اشباء پیچیده Introducing JavaScript Object Notation (JSON) مراجعه کنید.

|  |
| --- |
| var collection = {    2548: {      album: "Slippery When Wet",      artist: "Bon Jovi",      tracks: [        "Let It Rock",        "You Give Love a Bad Name"      ]    },    2468: {      album: "1999",      artist: "Prince",      tracks: [        "1999",        "Little Red Corvette"      ]    },    1245: {      artist: "Robert Palmer",      tracks: [ ]    },    5439: {      album: "ABBA Gold"    }  };  // فقط کد زیر را تعقیر دهید  function updateRecords(id, prop, value) {    return collection;  }  updateRecords(5439, "artist", "ABBA"); |

## تکرار با حلقه whileدر JavaScript

می توانید با استفاده از یک حلقه ، چندین بار یک کد را اجرا کنید. نوع اول حلقه ای که ما یاد خواهیم گرفت حلقه while نامیده می شود زیرا آن زمانی که یک شرایط خاص وجود دارد( true است) اجرا می شود و هنگامی که دیگر این شرط برقرار نباشد متوقف می شود.

|  |
| --- |
| var ourArray = [];  var i = 0;  while(i < 5) {    ourArray.push(i);    i++;  } |

در مثال کد بالا ، حلقهwhile، 5 بار اجرا می شود و اعداد 0 تا 4 را به ourArray اضافه می کند.

بیایید با اضافه کردن مقادیر به آرایه ، روی حلقه while تمرین کنیم.

تمرین 1-93: اعداد 5 تا 0 (شامل) را به ترتیب نزولی به myArray با استفاده از یک حلقه while اضافه کنید.

|  |
| --- |
| var myArray = []; |

## تکرار با حلقه forدر JavaScript

می توانید با استفاده از یک حلقه ، چندین بار یک کد را اجرا کنید. متداول‌ترین نوع حلقه جاوا اسکریپت یک حلقهfor نامیده می شود زیرا برای تعداد مشخصی از دفعات اجرا می شود.

حلقه های for با سه عبارت اختیاری که با یک سمیکولن(;) جدا می شوند اعلان می‌شوند:

for ([initialization]; [condition]; [final-expression])

دستور initializationفقط یک بار قبل از شروع حلقه اجرا می شود. معمولاً برای تعریف و تنظیم متغیر حلقه شما استفاده می شود.

عبارت condition در ابتدای هر تکرار حلقه ارزیابی می شود و تا زمانی که true ارزیابی شود ادامه خواهد یافت. هنگامی که condition در شروع تکرار false باشد ، اجرای حلقه متوقف می شود. این بدان معنی است که اگر condition از ابتدای شروع حلقه false شود ، حلقه شما هرگز اجرا نخواهد شد.

final-expression در پایان هر تکرار حلقه ، قبل از بررسی وضعیت بعدی اجرا می شود و معمولاً برای افزایش یا کاهش شمارنده حلقه شما استفاده می شود.

در مثال زیر با i = 0 شروع می کنیم و تا زمانی که شرط i <5 ، true است تکرا می‌کنیم. ما در هر تکرار حلقه 1 واحد به i با i++ به عنوان final-expression اضافه خواهیم کرد.

|  |
| --- |
| var ourArray = [];  for (var i = 0; i < 5; i++) {    ourArray.push(i);  } |

ourArray اکنون حاوی [0،1،2،3،4] خواهد بود.

تمرین 1-94: از یک حلقه برای اضافه کردن مقادیر 1 تا 5 را روی myArray استفاده کنید.

|  |
| --- |
| var myArray = []; |

## تکرار با اعداد فرد با حلقه for

حلقه for در هر بار اجرا مجبور نیستید یک بار تکرار کنید. با تغییر final-expression ما ، می توانیم حتی با اعداد زوج شمارش کنیم.

ما از i = 0 شروع می کنیم و تا زمانی که i <10 باشد حلقه را تکرار می کنیم. هر بار i با i+=2 دو برابر افزایش می‌یابد.

|  |
| --- |
| var ourArray = [];  for (var i = 0; i < 10; i += 2) {    ourArray.push(i);  } |

ourArray اکنون حاوی [0،2،4،6،8] خواهد بود. بیایید مقداردهی اولیه خود را تغییر دهیم تا بتوانیم با اعداد فرد شمارش کنیم.

تمرین 1-95: با استفاده از حلقه اعداد فرد از 1 تا 9 به myArray اضافه کنید.

|  |
| --- |
| var myArray = []; |

## شمارش به عقب با حلقه for

تا زمانی که بتوانیم شرایط مناسب را تعریف کنیم، یک حلقه for همچنین می تواند به عقب شمارش کند. برای شمارش به عقب دوتایی ، نیاز داریم که initialization ، condition و final-expressionخود را تغییر دهیم.

ما در i = 10 حلقه را شروع می کنیم تا زمانی که i> 0. ما در هر تکرار 2 را از i با i - = 2 کاهش می دهیم.

|  |
| --- |
| var ourArray = [];  for (var i = 10; i > 0; i -= 2) {    ourArray.push(i);  } |

ourArray اکنون حاوی [10،8،6،4،2] خواهد بود. بیایید initializationو final-expression خود را تغییر دهیم تا بتوانیم دوتا دوتا با اعداد فرد به عقب شمارش کنیم.

تمرین 1-96: با استفاده از حلقه for ، اعداد فرد از 9 تا 1 به myArray اضافه کنید.

|  |
| --- |
| var myArray = []; |

## پیمایش میان عناصر آرایه با یک حلقه for

یک کار رایج در JavaScript پیمایش محتوای یک آرایه است. یک راه برای انجام این کار با حلقه for است. این کد باعث می شود که هر عنصر از آرایه به کنسول برود:

|  |
| --- |
| var arr = [10, 9, 8, 7, 6];  for (var i = 0; i < arr.length; i++) {     console.log(arr[i]);  } |

به یاد داشته باشید که آرایه ها دارای ایندکس از صفر هستند ، به این معنی که آخرین ایندکس این آرایه length - 1است. شرط ما برای این حلقه i < arr.length است ، که حلقه را هنگامی که i برابر با length است متوقف می کند. در این حالت آخرین تکرار i === 4 است یعنی وقتی i برابر است با arr.length و خروجی 6 به کنسول می رود.

تمرین 1-97: یک متغیر total را اعلان و با 0 مقدار دهی اولیه کنید. از یک حلقه استفاده کنید تا مقدار هر عنصر از آرایه myArr را به total اضافه کنید.

|  |
| --- |
| var myArr = [ 2, 3, 4, 5, 6]; |

## حلقه‌های for تودرتو

اگر آرایه چندبعدی دارید ، می توانید از همان منطق مثل روش قبلی برای پیمایش کردن در هر آرایه و هر زیر آرایه استفاده کنید. به عنوان مثال:

|  |
| --- |
| var arr = [    [1,2], [3,4], [5,6]  ];  for (var i=0; i < arr.length; i++) {    for (var j=0; j < arr[i].length; j++) {      console.log(arr[i][j]);    }   } |

این هر یک از عناصر فرعی arr را در یک زمان یکجا به خروجی می‌فرستد. توجه داشته باشید که برای حلقه داخلی، ما .lengthاز arr[i] را بررسی می کنیم ، زیرا arr[i] خود یک آرایه است.

تمرین 1-98: تابع multiplyAll را طوری تغییر دهید که تمام اعداد موجود در زیر مجموعه arr را بازگرداند.

|  |
| --- |
| function multiplyAll(arr) {    var product = 1;  // فقط کد زیر این خط را تعقیر دهید   // فقط کد بالای این خط را تغقیر دهید    return product;  }  multiplyAll([[1,2],[3,4],[5,6,7]]); |

## تکرار با حلقه Do...While جاوااسکریپت

نوع بعدی حلقه‌ای که یاد خواهید گرفت ، do...while نامیده می‌شود .بخاطر این که چون ابتدا دستورات را یکبار در هر صورت اجرا می‌کند مهم نیست که چه چیزی باشد وسپس قسمت شرط while را برای true بودن ارزیابی می کند.

|  |
| --- |
| var ourArray = [];  var i = 0;  do {    ourArray.push(i);    i++;  } while (i < 5); |

مثال بالا مانند سایر انواع حلقه ها رفتار می کند ، و نتیجه آرایه شبیه به [0, 1, 2, 3, 4]خواهد بود. با این حال ، آنچه باعث می شود do...while که با سایر حلقه ها متفاوت باشد این است که وقتی اولین بررسی شرط با شکست مواجه شود ، چگونه رفتار می کند. بیایید این را در عمل مشاهده کنیم: در اینجا یک حلقه معمولی while است که کد را در حلقه تا زمانی که i <5 باشد اجرا می کند:

|  |
| --- |
| var ourArray = [];  var i = 5;  while (i < 5) {    ourArray.push(i);    i++;  } |

در این مثال مقدار ourArray را به یک آرایه خالی و مقدار i را با 5 مقدار دهی می کنیم. وقتی که ما حلقه while را اجرا می کنیم شرط false ارزیابی می شود زیرا i کمتر از 5 نیست، بنابراین کد داخل حلقه اجرا نمی‌شود. نتیجه این است که ourArray بدون هیچ مقدار افزوده به آن پایان می یابد ، و وقتی اجرای تمام کد های مثال بالا تمام شود ، همچنان [] خالی به نظر می رسد. حالا بیایید نگاهی به حلقه do…while بیاندازیم:

|  |
| --- |
| var ourArray = [];  var i = 5;  do {    ourArray.push(i);    i++;  } while (i < 5); |

در این حالت ، مقدار i را با 5 مقداردهی می کنیم ، دقیقاً مانند آنچه که با حلقه while انجام دادیم. وقتی به خط بعدی رسیدیم ، هیچ مشکلی برای ارزیابی وجود ندارد ، بنابراین به سراغ کد داخل بلوک{ } می رویم و آن را اجرا می کنیم. ما قبل از رسیدن به بررسی شرط ، یک عنصر واحد را به آرایه اضافه می کنیم و سپس i را یک واحد افزایش می‌دهیم. وقتی در آخر وضعیت i <5 را در آخرین خط ارزیابی می کنیم ، می بینیم که i اکنون 6 هست که بررسی شرط را با شکست مواجه می کند ، بنابراین از حلقه خارج می شود و تمام می‌شود. در پایان مثال بالا ، مقدار OurArray برابر است [5] است. اساساً ، یک حلقه do...while اطمینان حاصل می کند که کد داخل حلقه حداقل یک بار اجرا شود. بیایید با افزودن مقادیر به آرایه ، این کار را انجام دهیم.

تمرین 1-99: حلقه while کد را به یک حلقه do...while تغییر دهید که حلقه فقط عدد 10 را به myArray اضافه می‌کند،و وقتی که اجرای کد شما تمام شود i برابر با 11 خواهد بود.

|  |
| --- |
| var myArray = [];  var i = 10;  // فقط کد زیر این خط را تعقیر دهید  while (i < 5) {    myArray.push(i);    i++;  } |

## جایگزینی حلقه ها با استفاده از بازگشتی

بازگشتی مفهومی است که یک تابع را می توان از نظر خودش بیان کرد. برای درک این موضوع ، با فکر کردن در مورد کار زیر شروع کنید: اولین عناصر n یک آرایه را چند برابر کنید تا حاصل آن عناصر ایجاد شود. با استفاده از حلقه for می توانید اینگونه این کار را انجام دهید:

|  |
| --- |
| function multiply(arr, n) {      var product = 1;      for (var i = 0; i < n; i++) {          product \*= arr[i];      }      return product;    } |

توجه داشته باشید که multiply(arr, n) == multiply(arr, n - 1) \* arr[n - 1] . این بدان معناست که شما می توانید به نوبه خود multiply را بازنویسی کنید و هرگز نیازی به استفاده از یک حلقه نیست.

|  |
| --- |
| function multiply(arr, n) {      if (n <= 0) {        return 1;      } else {        return multiply(arr, n - 1) \* arr[n - 1];      }    } |

نسخه بازگشتی از multiply مانند این تجزیه می شود. در حالت پایه ، در جایی که n <= 0 است ، 1 بازمی گردد. برای مقادیر بزرگتراز n ، خود را صدا می کند ، اما با n - 1.

آن فراخوانی تابع به همان روش ارزیابی می شود ، multiply دوباره فراخوانی می شود تا n <= 0 شود. در این مرحله همه توابع می توانند برگردند و multiply اصلی جواب را برگرداند.

توجه: توابع بازگشتی وقتی که بدون فراخوانی مجدد تابع باز می گردانند ، باید یک حالت پایه داشته باشند (در این مثال ، هنگامی که n <= 0) ، در غیر این صورت هرگز نمی توانند اجرای را به پایان برسانند.

تمرین 1-100: یک تابع بازگشتی ، sum(arr, n) بنویسید ، که مجموع عناصر n اول یک آرایه را برمی گرداند.

|  |
| --- |
| function sum(arr, n) {    // فقط کد زیر این خط را تعقیر دهید    // فقط کد بالی این خط را تعقیر دهید  } |

## تمرین 1-101: جستجوی پروفایل

ما آرایه ای از اشیاء داریم که افراد مختلف درون لیست مخاطبین را نشان می دهد.

یک تابع lookUpProfile که نام و ویژگی(prop)که برای شما از قبل نوشته شده است را به عنوان آرگومان می‌گیرد. این تابع باید بررسی کند که آیا نام یک نام مخاطب واقعی است یا خیر و آن ویژگی(prop) داده شده ویژگی آن مخاطب است. اگر هر دو true هستند ، "مقدار" آن ویژگی را برگردانید. اگر نام با هیچ مخاطبی مطابقت ندارد ، " No such contact" را برگردانید. اگر prop با هیچ ویژگی معتبر مخاطبی که با نام مطابقت دارد مطابقت ندارد ، " No such property" را برگردانید.

|  |
| --- |
| var contacts = [      {          "firstName": "Akira",          "lastName": "Laine",          "number": "0543236543",          "likes": ["Pizza", "Coding", "Brownie Points"]      },      {          "firstName": "Harry",          "lastName": "Potter",          "number": "0994372684",          "likes": ["Hogwarts", "Magic", "Hagrid"]      },      {          "firstName": "Sherlock",          "lastName": "Holmes",          "number": "0487345643",          "likes": ["Intriguing Cases", "Violin"]      },      {          "firstName": "Kristian",          "lastName": "Vos",          "number": "unknown",          "likes": ["JavaScript", "Gaming", "Foxes"]      }  ];  function lookUpProfile(name, prop){    // فقط کد زیر این خط را تعقیر دهید    // فقط کد بالای این خط را تعقیر دهید  }  lookUpProfile("Akira", "likes"); |

## تولید توابع تصادفی با javaScript

اعداد تصادفی برای ایجاد رفتار تصادفی مفید هستند. جاوا اسکریپت یک تابع Math.random() دارد که یک عدد اعشار تصادفی بین 0 (خودش هست) تا 1 (خودش نیست) تولید می کند. بنابراین Math.random() می‌تواند 0 را بازگرداند اما هرگز 1 را برنمی گرداند.

توجه داشته باشیدکه مانند ذخیره کردن مقادیر با عملگر مساوی، تمام فراخوانی‌های تابعی قبل از اجرای return ارزیابی می‌شوند ، بنابراین می توانیم مقدار تابع Math.random() را برگردانیم.

تمرین 1-102: تابع randomFraction را برای بازگرداندن یک عدد تصادفی به جای 0 تعقیر دهید.

|  |
| --- |
| function randomFraction() {    // فقط کد زیر این خط را تعقیر دهی    return 0;    // فقط کد بالای این خط را تعقیر دهید  } |

## تولید اعداد تصادفی کامل با javaScript

بسیار عالی است که می توانیم اعداد اعشاری تصادفی تولید کنیم ، اما اگر از آن برای تولید اعداد صحیح تصادفی استفاده کنیم ، حتی مفیدتر است.

1. از Math.random() برای ایجاد یک عدد دهدهی تصادفی استفاده کنید.
2. آن رقم تصادفی را در 20 ضرب کنید.
3. از یک تابع دیگر ، Math.floor() استفاده کنید تا عدد را به نزدیکترین عدد پایین خود گرد کنید.

به یاد داشته باشید که Math.random ()هرگز نمی تواند عدد 1 را برگرداند و چون در حال گرد کردن به پایین هستیم ، غیرممکن است که بتوانیم 20 را بدست آوریم. این تکنیک بین 0 تا 19 عدد کاملی به ما می دهد.

همه چیز را با هم جمع می کنیم ، این همان چیزی است که کد ما شبیه به آن است:

Math.floor(Math.random() \* 20);

ما در حال فراخوانی Math.random() هستیم و نتیجه را در 20 ضرب می کنیم ، سپس مقدار را به تابع Math.floor() ارسال می کنیم تا مقدار را به نزدیکترین عدد پایین گرد کنیم.

تمرین 1-103: از این تکنیک برای تولید و بازگرداندن عدد تصادفی بین 0 تا 9 استفاده کنید.

|  |
| --- |
| function randomWholeNum() {    return Math.random();  } |

## تولید اعداد تصادفی کامل در یک محدوده خاص

به جای تولید یک عدد تصادفی بین صفر و یک عدد مشخص مانند قبل ، می توانیم یک عدد تصادفی تولید کنیم که در محدوده دو عدد خاص قرار می گیرد.

برای انجام این کار، ما یک حداقل عدد min و یک حداکثر عدد max تعریف خواهیم کرد. در اینجا یک فرمول است که از آن استفاده خواهیم کرد. کد را بخوانید و سعی کنید درک کنید که چه کاری انجام می‌دهد:

Math.floor(Math.random() \* (max - min + 1)) + min

تمرین 1-104: تابعی بنام randomRange ایجاد کنید که myMin و myMax دامنه شما را بگیرد و اعداد تصادفی بزرگتر یا مساوی با myMin ، و کمتر یا مساوی myMax ، را برگرداند.

|  |
| --- |
| function randomRange(myMin, myMax) {    return 0;} |

## استفاده از تابع parseInt

تابع parseInt() یک رشته را تجزیه می کند و یک عدد صحیح را برمی گرداند. در اینجا مثالی آورده شده است:

var a = parseInt("007");

تابع فوق رشته "007" را به عدد صحیح 7 تبدیل می کند. اگر اولین کاراکتر رشته نتواند به یک عدد تبدیل شود ، آنگاه NaN را برمی گرداند.

تمرین 1-105: از تابع parseInt() در تابع convertToInteger برای تبدیل رشته ورودی str به یک عدد صحیح استفاده کنید، و آن را برگردانید.

|  |
| --- |
| function convertToInteger(str) {  }  convertToInteger("56"); |

**حل:**

|  |
| --- |
|  |

var a = parseInt("007");

## استفاده از تابع parseInt با یک Radix

تابع parseInt() یک رشته را تجزیه می کند و یک عدد صحیح را برمی گرداند. یک آرگومان دوم برای Radix می گیرد ، که پایه عدد را در رشته مشخص می کند. Radix می تواند یک عدد صحیح بین 2 تا 36 باشد.

فراخوانی تابع اینگونه به نظر می‌رسد:

parseInt(string, radix);

و در اینجا مثالی وجود دارد:

var a = parseInt("11", 2);

متغیر radix می گوید "11" در سیستم دودویی در پایه 2 است. این مثال رشته "11" را به عدد 3 تبدیل می‌کند.

تمرین 1-106: در تابع convertToInteger از parseInt()استفاده کنید ، که یک عدد باینری را به یک عدد صحیح تبدیل می‌کند و آن را برمی گرداند.

|  |
| --- |
| function convertToInteger(str) {  }  convertToInteger("10011"); |

## استفاده از عملگر شرطی (سه گانه)

عملگر شرطی ، که به آن عملگر سه گانه نیز گفته می شود ، می تواند به عنوان یک دستور if-else یک خطی استفاده شود. نحو آن اینگونه است:

condition ? statement-if-true : statement-if-false;

تابع زیر از یک دستور if-else برای بررسی شر ط استفاده می کند:

|  |
| --- |
| function findGreater(a, b) {    if(a > b) {      return "a is greater";    }    else {      return "b is greater";    }  } |

این دستور را می توان دوباره با استفاده از عملگر شرطی نوشت:

|  |
| --- |
| function findGreater(a, b) {    return a > b ? "a is greater" : "b is greater";} |

تمرین 1-107: از عملکرد شرطی در تابع checkEqual استفاده کنید تا بررسی کنید که آیا دو عدد مساوی هستند یا خیر. تابع باید یا "Equal" یا "Not Equal" را بازگرداند.

|  |
| --- |
| function checkEqual(a, b) {  }  checkEqual(1, 2); |

## استفاده از عملگرهای چندشرطی (سه گانه)

در چالش قبلی ، شما از یک عملگر شرطی منفرد استفاده کرده اید.شما همچنین می توانید آنها را به هم متصل کنید تا چندین حالت را بررسی کنید. تابع زیر از عبارات if و else برای بررسی چندین شرط استفاده می کند:

|  |
| --- |
| function findGreaterOrEqual(a, b) {    if (a === b) {      return "a and b are equal";    }    else if (a > b) {      return "a is greater";    }    else {      return "b is greater";    }  } |

تابع بالا را می توان با استفاده از چندین عملگر شرطی دوباره نوشت:

|  |
| --- |
| function findGreaterOrEqual(a, b) {    return (a === b) ? "a and b are equal"      : (a > b) ? "a is greater"      : "b is greater";  } |

بهترین قالب برای قالب بندی چندین عملگر شرطی به گونه ای است که مطابق شکل بالا ، هر شرط در یک خط جداگانه قرار دارد. استفاده از چندین عملگر شرطی بدون قرار دادن آن ها درجای مناسب ممکن است خواندن را برای شما سخت کند. مثلا:

|  |
| --- |
| function findGreaterOrEqual(a, b) {    return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";  } |

تمرین 1-108: در تابع checkSign ، از چندین عملگر شرطی استفاده کنید - از فرمت پیشنهادی که در findGreaterOrEqual استفاده شد پیروی کنید - برای بررسی اینکه آیا یک عدد مثبت ، منفی یا صفر است. تابع باید "positive" ، "negative" یا "zero" را برگرداند.

|  |
| --- |
| function checkSign(num) {  }  checkSign(10); |

## استفاده از بازگشتی برای ایجاد یک شمارش معکوس

در یکی از چالش های قبل ، شما یاد گرفتید که چگونه از بازگشتی برای جایگزینی حلقه for استفاده کنید. حال ، به یک تابع پیچیده تر می پردازیم که آرایه ای از اعداد صحیح متوالی را با شروع از 1 که به تابع ارسال شده را برمی‌گرداند.

همانطور که در چالش قبلی ذکر شد ، یک مورد بایه وجود خواهد داشت. مورد پایه هنگامی که دیگر نیازی به فراخوانی اش نباشد ، به تابع بازگشتی اطلاع می‌دهد. این یک مورد ساده است که در آن مقدار بازگشتی قبلاً مشخص است. همچنین یک فراخوانی بازگشتی وجود دارد که تابع اصلی را با آرگومان های مختلف اجرا می‌کند. اگر تابع به درستی نوشته شده باشد ، در نهایت به پایه رسیده می شود.

به عنوان مثال ، به شما می گویند یک تابع بازگشتی بنویسید که آرایه ای شامل اعداد 1 تا n را برگرداند. این تابع نیاز به پذیرش یک آرگومان ، n دارد ، که شماره آخر را نشان می‌دهد. سپس باید مقدار خود را با مقادیر تدریجی کوچکتر n تا زمان رسیدن به 1 فراخواند. شما می توانید این تابع را به شکل زیر بنویسید:

|  |
| --- |
| function countup(n) {    if (n < 1) {      return [];    } else {      const countArray = countup(n - 1);      countArray.push(n);      return countArray;    }  }  console.log(countup(5)); // [ 1, 2, 3, 4, 5 ] |

در ابتدا ، به نظر می رسد که کارآمد است زیرا مقدار n کاهش می یابد ، اما مقادیر موجود در آرایه نهایی در حال افزایش است. زیرا اضافه کردن به انتها ، پس از بازگشت فراخوانی بازگشتی اتفاق می‌افتد. در نقطه ای که n به آرایه منتقل می شود ،(countup(n - 1 از قبل ارزیابی شده و[1, 2, ..., n - 1]. بازگردانده شده است

تمرین 1-109: ما تابعی به نام countdown با یک پارامتر (n) تعریف کرده ایم. این تابع باید از بازگشتی برای بازگشت آرایه ای شامل اعداد صحیح n از 1 بر اساس پارامتر n استفاده کند. اگر تابع با عدد کمتر از 1 فراخوانی شود ، تابع باید یک آرایه خالی را برگرداند. به عنوان مثال ، فراخوانی این تابع با n = 5 باید آرایه [5, 4, 3, 2, 1] را بازگرداند. تابع شما باید برای فراخوانی خودش از بازگشتی استفاده کند و از هر نوع حلقه ای نباید استفاده کند.

|  |
| --- |
| function countdown(n){    return;  } |

## استفاده از بازگشتی برای ایجاد طیف وسیعی از اعداد

در ادامه چالش قبلی ، ما فرصتی دیگر برای ایجاد یک تابع بازگشتی برای حل یک مشکل به شما می‌دهیم.

تمرین 1-110: ما تابعی به نام rangeOfNumbers با دو پارامتر تعریف کرده ایم. این تابع باید آرایه ای از اعداد صحیح را برگرداند که از یک عدد نشان داده شده است توسط پارامتر startNum شروع می شود و با یک عدد نشان داده شده توسط پارامتر endNum به پایان می رسد. عدد شروع همیشه کمتر یا مساوی با عدد پایان خواهد بود. تابع شما باید از بازگشتی بوسیله فراخوانی خودش استفاده کند و از هیچ حلقه ای استفاده نکند. همچنین باید برای مواردی که هم startNum و هم endNum یکسان باشند هم کار کند.

|  |
| --- |
| function rangeOfNumbers(startNum, endNum) {    return [];  }; |

# 2. ES6

## آشنایی با چالش های ES6

ECMAScript یک نسخه استاندارد از JavaScript با هدف یکی کردن مشخصات و ویژگی های زبان است. از آنجا که همه مرورگرهای اصلی و زمان اجرای JavaScript از این مشخصات پیروی می کنند ، اصطلاح ECMAScript با اصطلاح JavaScript قابل تعویض است.

بسیاری از چالش های مربوط به freeCodeCamp از مشخصات ECMAScript 5 (ES5) این زبان استفاده می کنند ، که در سال 2009 نهایی شد. اما JavaScript یک زبان برنامه نویسی در حال تحول است. با افزودن ویژگی ها و اصلاحات ، نسخه های جدید این زبان برای استفاده توسعه دهندگان منتشر می شود.

اما جاوا اسکریپت یک زبان برنامه نویسی در حال حاضر استول است. با افزودن ویژگی ها و اصلاحات ، نسخه های جدید این زبان را برای استفاده از دهکده امکان پذیر می سازد.

جدیدترین نسخه استاندارد شده ECMAScript 6 (ES6) نام دارد که در سال 2015 منتشر شد. این نسخه جدید زبان چندین ویژگی قدرتمند را اضافه می کند که در این بخش از چالش ها قرار خواهد گرفت ، از جمله:

* توابع Arrow
* کلاس‌ها
* ماژول‌ها
* Promise ها
* Generator ها
* Let و const

توجه: همه مرورگرها از ویژگی های ES6 پشتیبانی نمی کنند. اگر از ES6 در پروژه های خود استفاده می کنید ، ممکن است لازم باشد برای تبدیل کد ES6 خود به ES5 تا زمانی که مرورگرها از ES6 پشتیبانی کنند ، از یک برنامه (transpiler) استفاده کنید.

## تفاوت بین کلمات کلیدی var و let

یکی از بزرگترین مشکلات مربوط به اعلامن متغیرها با کلید واژه var این است که می توانید بدون خطا تعریف متغیرها را بازنویسی کنید.

|  |
| --- |
| var camper = 'James';  var camper = 'David';  console.log(camper);  // logs 'David' |

همانطور که در کد بالا مشاهده می کنید ، متغیر camper در ابتدا به عنوان James اعلان می شود و پس از آن overridden شده تا David باشد. در یک برنامه کوچک ، شما ممکن است به این نوع مشکل مواجه نشوید ، اما هنگامی که کد شما بزرگتر می شود ، ممکن است به طور تصادفی متغیری را که قصد بازنویسی آن را ندارید بازنویسی کنید. از آنجا که این رفتار خطایی ایجاد نمی کند ، جستجو و رفع اشکالات دشوارتر می شود. یک کلمه کلیدی جدید به نام let در ES6 برای حل این مشکل احتمالی با کلید واژه var معرفی شده است. اگر می‌خواهید var را با let در اعلان های متغیر کد بالا جایگزین کنید ، نتیجه آن خطایی خواهد بود.

|  |
| --- |
| let camper = 'James';  let camper = 'David'; // throws an error |

این خطا در کنسول مرورگر شما قابل مشاهده است. بنابراین برخلاف var ، هنگام استفاده از let ، یک متغیر با همین نام فقط یک بار می تواند اعلان شود. توجه داشته باشید "use strict".

این حالت Strict Mode را ممکن می کند ، که خطاهای رمزگذاری متداول و اقدامات "ناامن" را می‌گیرد. حالتstrict mode در واقع حالتی است که در آن کدها با شرایط سخت گیرانه تری اجرا می شوند. برای مثال:

|  |
| --- |
| "use strict";  x = 3.14; // throws an error because x is not declared |

### تمرین2-1: کد را به روز کنید تا فقط از کلمه کلیدی let استفاده کند.

|  |
| --- |
| var catName;  var quote;  function catTalk() {    "use strict";    catName = "Oliver";    quote = catName + " says Meow!";  }  catTalk(); |

## مقایسه دامنه های کلمات کلیدی var و let

وقتی متغیر را با کلمه کلیدی var تعریف کنید ، به صورت سراسری اعلان می‌شود و اگر داخل یک تابع تعریف شود محلی اعلان می شود.

کلمه کلیدی let به طور مشابه رفتار می کند اما با برخی ویژگی های اضافی. وقتی متغیر را با کلمه کلیدی let در داخل یک بلوک ، دستور اعلان می کنید ، دامنه آن محدود به آن بخش ، دستور یا عبارت است. برای مثال:

|  |
| --- |
| var numArray = [];  for (var i = 0; i < 3; i++) {    numArray.push(i);  }  console.log(numArray);  // returns [0, 1, 2]  console.log(i);  // returns 3 |

با کلمه کلیدی var ، i به صورت سراسری اعلان می شود. بنابراین وقتی i++ اجرا شد ، متغیر سراسری i را بروزرسانی می کند. این کد مشابه موارد زیر است:

|  |
| --- |
| var printNumTwo;  for (var i = 0; i < 3; i++) {    if (i === 2) {      printNumTwo = function() {        return i;      };    }  }  console.log(printNumTwo());  // returns 3 |

همانطور که مشاهده می کنید ، printNumTwo()، 3 را و نه 2 را چاپ می کند. دلیل این است که مقدار اختصاص داده شده به i به روز شده و printNumTwo() i سراسری را بازمی گرداند و نه مقداری را که هنگام ایجاد تابع در حلقه ایجاد شده است. کلمه کلیدی let از این رفتار پیروی نمی کند:

|  |
| --- |
| 'use strict';  let printNumTwo;  for (let i = 0; i < 3; i++) {    if (i === 2) {      printNumTwo = function() {        return i;      };    }  }  console.log(printNumTwo());  // returns 2  console.log(i);  // returns "i is not defined" |

i تعریف نشده زیرا در محدوده سراسری اعلان نشده است. این فقط در دستور حلقه اعلان می شود. printNumTwo() مقدار صحیح را برگرداند زیرا سه متغیر مختلف i با مقادیر منحصر به فرد (0 ، 1 و 2) توسط کلمه کلیدی let در دستور حلقه ایجاد شده اند.

تمرین2-2: كد را برطرف كنيد تا i را كه در دستور if اعلان شده است يك متغير جداگانه از i که در خط اول تابع اعلان شد باشد. مطمئن شوید که از کلمه کلیدی var در هر کجای کد استفاده نمی کنید.

این تمرین به منظور نشان دادن تفاوت بین var و let در چگونگی اختصاص دامنه به متغیر تعریف شده طراحی شده است. هنگام برنامه نویسی تابعی مشابه تابع مورد استفاده در این تمرین ، اغلب بهتر است از اسامی متغیر مختلف استفاده شود تا از سردرگمی جلوگیری شود.

|  |
| --- |
| function checkScope() {    'use strict';    var i = 'function scope';    if (true) {      i = 'block scope';      console.log('Block scope i is: ', i);    }    console.log('Function scope i is: ', i);    return i;  } |

## تعریف متغیر فقط خواندنی با کلمه کلیدی const

کلمه کلیدی let تنها روش جدید برای اعلان متغیرها نیست. در ES6 همچنین می توانید متغیرها را با استفاده از کلید واژه const اعلان کنید.

const دارای تمام ویژگی های عالی است که let دارد، با امکان اضافی که متغیرهای اعلان شده با استفاده از const فقط خواندنی هستند. آنها یک مقدار ثابت هستند ، به این معنی که وقتی یک متغیر با const اختصاص داده شود ، امکان تغییر مجدد وجود ندارد.

|  |
| --- |
| "use strict";  const FAV\_PET = "Cats";  FAV\_PET = "Dogs"; // returns erro |

همانطور که مشاهده می کنید ، تلاش برای تنظیم مجدد متغیر اعلان شده با const خطایی را به همراه خواهد داشت. شما همیشه باید متغیرهایی را که نمی خواهید مجدداً مقداردهی کنید را با استفاده از کلید واژه const اعلان کنید. این امر به شما کمک می کند که اگر به طور اتفاقی سعی کنید مجدداً یک متغیر را تغییر دهید که ثابت باشد ،ثابت بماند. یک روش معمول در نامیدن ثابت ها استفاده از همه حروف بزرگ است ، با کلمات جدا شده توسط underscore (\_).

توجه: توسعه دهندگان معمولاً از شناسه های متغیرحروف بزرگ برای مقادیر تغییر ناپذیر و حروف کوچک یا camelCase برای مقادیر قابل تغییر (اشیاء و آرایه ها) استفاده می کنند. در چالش بعدی ، نمونه‌ای از شناسه متغیر کوچک را برای استفاده از آرایه مشاهده خواهید کرد.

تمرین2-3: کد را تغییر دهید تا تمام متغیرها با استفاده از let یا const اعلام شوند. هنگامی که می خواهید متغیر تغییر کند از let ، و از متغیر const برای زمانی که می‌خواهید ثابت بماند استفاده کنید. همچنین ، متغیرهای تعریف شده به صورت const را با توجه به تمرینات جاری تعقیر نام دهید، ثابت ها همه باید با حروف بزرگ نوشته شوند.

|  |
| --- |
| function printManyTimes(str) {    "use strict";    // فقط کد زیر این خط را تعقیر دهید    var sentence = str + " is cool!";    for (var i = 0; i < str.length; i+=2) {      console.log(sentence);    }    // فقط کد بالای این خط را تعقیر دهید  }  printManyTimes("freeCodeCamp"); |

## تعقیرآرایه تعریف شده با const

دستور const کاربرد بسیار زیادی در JavaScript مدرن دارد. برخی از توسعه دهندگان ترجیح می دهند تمام متغیرهای خود را با استفاده از const بصورت پیش فرض تخصیص دهند ، مگر اینکه بدانند که نیاز به تنظیم مجدد مقدار دارند. فقط در این حالت ، آنها از let استفاده می کنند.

با این حال ، این مهم است که درک کنیم که اشیاء (از جمله آرایه ها و توابع) اختصاص داده شده به یک متغیر با استفاده از const هنوز قابل تغییر هستند. استفاده از دستور const فقط از انتساب مجدد شناسه متغیر جلوگیری می کند.

|  |
| --- |
| editInPlace();  "use strict";  const s = [5, 6, 7];  s = [1, 2, 3]; // throws error, trying to assign a const  s[2] = 45; // works just as it would with an array declared with var or let  console.log(s); // returns [5, 6, 45] |

همانطور که می بینید ، می توانید خود شی [5, 6, 7] را تعقیر دهید و متغیر s همچنان به آرایه تغییر یافته [5, 6, 45] اشاره خواهد کرد. مانند همه آرایه ها ، عناصر آرایه در s قابل تغییر هستند ، اما به دلیل استفاده از const ، نمی توانید از شناسه متغیر s استفاده کنید تا با استفاده از عملگر تخصیص(=) ، به آرایه های مختلفی اشاره کنید.

تمرین2-4: آرایه به صورت const s = [5, 7, 2] تعریف شده است. با استفاده از عناصر مختلف تخصیص، آرایه را به [2, 5, 7] تغییر دهید.

|  |
| --- |
| const s = [5, 7, 2];  function editInPlace() {    'use strict'; |

## جلوگیری از تعقیر شی

همانطور که در چالش قبلی مشاهده شد ، کلمه کلیدی Const به تنهایی از داده های شما در مقابل تعقیر محافظت نمی‌کند. برای اطمینان از عدم تغییر داده های شما ، JavaScript تابع Object.freeze برای جلوگیری از تعقیر داده فراهم می کند.

پس از یخ زدن شی ، دیگر نمی توانید خواص آن را اضافه کنید ، به روز کنید یا حذف کنید. هرگونه تلاش برای تغییر شی بدون خطا رد می شود.

|  |
| --- |
| let obj = {    name:"FreeCodeCamp",    review:"Awesome"  };  Object.freeze(obj);  obj.review = "bad"; // will be ignored. Mutation not allowed  obj.newProp = "Test"; // will be ignored. Mutation not allowed  console.log(obj);  // { name: "FreeCodeCamp", review:"Awesome"} |

تمرین2-5: در این چالش می خواهیم از Object.freeze استفاده کنید تا از تغییر ثابت ریاضی جلوگیری شود. شما باید شیء MATH\_CONSTANTS را freeze کنید تا کسی نتواند مقدار PI را تعقیر دهد ، ویژگی ها را اضافه یا حذف کند.

|  |
| --- |
| function freezeObj() {    'use strict';    const MATH\_CONSTANTS = {      PI: 3.14    };  // فقط کد زیر این خط را تعقیر دهید    // فقط کد بالای این خط را تعقیر دهید    try {      MATH\_CONSTANTS.PI = 99;    } catch(ex) {      console.log(ex);    }    return MATH\_CONSTANTS.PI;  }  const PI = freezeObj(); |

## استفاده از توابع Arrow برای نوشتن توابع ناشناس کوتاه‌ تر

در JavaScript ، ما اغلب نیازی به نام‌گذاری توابع خود نداریم ، به خصوص هنگام ارسال یک تابع به عنوان آرگومان به یک تابع دیگر. در عوض ، ما توابع درون خطی ایجاد می کنیم. نیازی به نامگذاری این توابع نیست زیرا ما از آنها در هیچ کجای دیگر استفاده نمی کنیم. برای رسیدن به این هدف ، ما اغلب از نحو زیر استفاده می کنیم:

|  |
| --- |
| const myFunc = function() {    const myVar = "value";    return myVar;  } |

ES6 نحوی را به ما ارائه می دهد تا مجبور نباشیم توابع ناشناس را از این طریق بنویسیم. در عوض ، می توانید از نحو تابع **arrow** استفاده کنید:

|  |
| --- |
| const myFunc = () => {    const myVar = "value";    return myVar;  } |

هنگامی که بدنه تابع وجود ندارد و فقط یک مقدار برگشتی وجود دارد ، نحو تابع Arrow به شما امکان می دهد تا کلمه کلیدی را فراموش نکنید و همچنین براکت های اطراف کد را حذف کنید. این به ساده کردن توابع کوچکتر در دستورات یک خطی کمک می کند:

|  |
| --- |
| const myFunc = () => "value"; |

تمرین2-6: تابع اختصاص داده شده به متغیر magic را بازنویسی کنید که new Date() را برای استفاده از نحو تابع Arrow باز می گرداند. همچنین ، اطمینان حاصل کنید که هیچ چیز با استفاده از کلید واژه var تعریف نشده است.

|  |
| --- |
| var magic = function() {    "use strict";    return new Date();  }; |

## نوشتن توابع Arrow با پارامترها

درست مانند یک تابع معمولی ، می توانید آرگومان‌ها را به یک تابع Arrow ارسال کنید.

|  |
| --- |
| // doubles input value and returns it  const doubler = (item) => item \* 2; |

دقیقاً مانند یک تابع معمولی ، اگر یک تابعArrow دارای یک آرگومان واحد باشد ، می توانید پرانتزهایی را که آن را محصور می کند ، حذف کنید.

|  |
| --- |
| // the same function, without the argument parentheses  const doubler = item => item \* 2; |

می توان بیش از یک آرگومان را به یک تابع Arrow ارسال کرد.

|  |
| --- |
| // multiplies the first input value by the second and returns it  const multiplier = (item, multi) => item \* multi; |

تمرین2-7: تابع myConcat را که به محتویات arr2 به arr1 اضافه می‌کند را بازنویسی کنید تا این تابع از نحو تابع Arrow استفاده کند.

|  |
| --- |
| var myConcat = function(arr1, arr2) {    "use strict";    return arr1.concat(arr2);  };  console.log(myConcat([1, 2], [3, 4, 5])); |

## تنظیم پارامتر‌های پیش‌فرض برای تابع خود

به منظور کمک به ما در ایجاد توابع انعطاف پذیرتر ، ES6 پارامترهای پیش فرض را برای توابع معرفی می کند. این کد را بررسی کنید:

|  |
| --- |
| const greeting = (name = "Anonymous") => "Hello " + name;  console.log(greeting("John")); // Hello John  console.log(greeting()); // Hello Anonymous |

پارامتر پیش فرض زمانی استفاده که آرگومان مشخص نشده باشد (تعریف نشده است). همانطور که در مثال بالا مشاهده می کنید ، نام پارامتر وقتی مقداری برای پارامتر ارائه نمی دهید ، مقدار پیش فرض "Anonymous" را دریافت می کند. می توانید مقادیر پیش فرض را برای هر پارامتر که می خواهید اضافه کنید.

تمرین2-8: تابع increment را با اضافه کردن پارامترهای پیش فرض اصلاح کنید تا در صورت مشخص نبودن value ، 1 به number اضافه شود.

|  |
| --- |
| const increment = (number, value) => number + value; |

## استفاده از پارامتر Rest با پارامتر های تابع

به منظور کمک به ما در ایجاد توابع انعطاف پذیرتر ، ES6 پارامتر Rest را برای پارامترهای تابع معرفی می کند. با پارامتر Rest ، می توانید توابعی ایجاد کنید که تعداد متغیرهای آرگومان را در بر می گیرد. این آرگومان‌ها در آرایه ای ذخیره می شوند که بعداً از داخل تابع قابل دستیابی است. این کد را بررسی کنید:

|  |
| --- |
| function howMany(...args) {    return "You have passed " + args.length + " arguments.";  }  console.log(howMany(0, 1, 2)); // You have passed 3 arguments.  console.log(howMany("string", null, [1, 2, 3], { })); // You have passed 4 arguments. |

پارامتر Rest نیاز به بررسی آرایه args را برطرف می کند و به ما امکان می دهد تا map()،filter() و reduce()را روی پارامتر‌ها اعمال کنیم.

تمرین2-9: تابع sum را با استفاده از پارامتر Rest اصلاح کنید به گونه ای که تابعsum قادر به گرفتن هر تعداد آرگومان باشد و جمع آنها را برگرداند.

|  |
| --- |
| const sum = (x, y, z) => {    const args = [x, y, z];    return args.reduce((a, b) => a + b, 0);  } |

## استفاده از عملگرSpread برای ارزیابی آرابه ها در محل

ES6 عملگر Spread را معرفی می کند ، که به ما امکان می دهد آرایه ها و عبارات دیگر را در مکان‌‌هایی که انتظار می رود چندین پارامتر یا عنصر داشته باشیم ، گسترش دهیم. کد ES5 زیر برای محاسبه حداکثر مقدار در یک آرایه از apply() استفاده می کند:

|  |
| --- |
| var arr = [6, 89, 3, 45];  var maximus = Math.max.apply(null, arr); // returns 89 |

ما مجبور شدیم از Math.max.apply(null, arr) استفاده کنیم زیرا Math.max(arr)، NaN را برمی‌گرداند. Math.max() آرگومان‌های جداشده با کاما را دارد ، اما یک آرایه نیست. عملگر Spread این نحو را برای خواندن و نگهداری بهتر می کند.

|  |
| --- |
| const arr = [6, 89, 3, 45];  const maximus = Math.max(...arr); // returns 8 |

... arr یک آرایه باز نشده را باز می گرداند. به عبارت دیگر ، آرایه را گسترش می دهد. با این حال ، عملگر گسترش فقط در یک مکان کار می کند ، مانند در یک آرگومان یک تابع یا در یک آرایه لیترال. کد زیر کار نخواهد کرد:

|  |
| --- |
| const spreaded = ...arr; // will throw a syntax error |

تمرین2-10: با استفاده از عملگر spread ، تمام محتویات arr1 را در یک آرایه دیگر arr2 کپی کنید.

|  |
| --- |
| const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];  let arr2;  arr2 = [];  console.log(arr2); |

## استفاده از Destructuring Assignment برای استخراج مقدار از شی

برای مرتب سازی منظم مقادیر گرفته شده به طور مستقیم از یک شی، Destructuring Assignmenنحو ویژه ای است که در ES6 معرفی شده است. کد ES5 زیر را در نظر بگیرید:

|  |
| --- |
| onst user = { name: 'John Doe', age: 34 };  const name = user.name; // name = 'John Doe'  const age = user.age; // age = 34 |

در اینجا دستور مقداردهی معادل با استفاده از نحو Destructuring ، ES6 آورده شده است:

|  |
| --- |
| const { name, age } = user;  // name = 'John Doe', age = 34 |

در اینجا متغیرهای name و age ایجاد می شوند و مقادیر مربوط به آنها را از شیء user تخصیص می دهند. شما می توانید ببینید که این چقدر تمیزتر است.

شما می توانید هر چه قدر می‌خواهید چه مقدار کم وجه مقدارهای خیلی زیاد از شی مورد نظر خود استخراج کنید.

تمرین2-11: دو انتساب را با یک Destructuring معادل جایگزین کنید. هنوز هم باید به متغیرهای today و tomorrow مقادیر today و tomorrow را از شی HIGH\_TEMPERATURES اختصاص دهید.

|  |
| --- |
| const HIGH\_TEMPERATURES = {    yesterday: 75,    today: 77,    tomorrow: 80  };  const today = HIGH\_TEMPERATURES.today;  const tomorrow = HIGH\_TEMPERATURES.tomorrow; |

## استفاده از Destructuring Assignment برای انتساب متغیرها از اشیا

Destructuring به شما امکان می دهد هنگام استخراج مقادیر ، یک نام متغیر جدید اختصاص دهید. می توانید این کار را با قرار دادن نام جدید بعد از کولن هنگام اختصاص مقدار انجام دهید. با استفاده از همان شیء از مثال آخر:

|  |
| --- |
| const user = { name: 'John Doe', age: 34 }; |

در اینجا چگونه شما می توانید نام متغیرهای جدید را در انتساب بیاورید :

|  |
| --- |
| const { name: userName, age: userAge } = user;  // userName = 'John Doe', userAge = 34 |

شما ممکن است آن را به عنوان "مقدار user.name را بدست آورید و آن را به متغیر جدیدی به نام userNameو غیره اختصاص دهید".

تمرین2-12: دو انتساب را با یک Destructuring معادل جایگزین کنید. هنوز هم باید به متغیرهای highToday و highTomorrow مقادیر today و tomorrow را از شی HIGH\_TEMPERATURES اختصاص دهید.

|  |
| --- |
| const HIGH\_TEMPERATURES = {    yesterday: 75,    today: 77,    tomorrow: 80  };  const highToday = HIGH\_TEMPERATURES.today;  const highTomorrow = HIGH\_TEMPERATURES.tomorrow; |

## استفاده از Destructuring Assignment برای انتساب متغیرها از اشیا تودرتو

مانند دو درس قبلی می توانید برای از تعقیر ساختار مقادیر اشیاء تو در تو ، از همان اصول استفاده کنید. با استفاده از یک شی مشابه نمونه های قبلی:

|  |
| --- |
| const user = {    johnDoe: {      age: 34,      email: 'johnDoe@freeCodeCamp.com'    }  }; |

در اینجا نحوه استخراج مقادیر خاصیت های شیء و اختصاص آنها به متغیرهایی با همین نام آورده شده است:

|  |
| --- |
| const { johnDoe: { age, email }} = user; |

و در اینجا چگونه می توانید مقادیر ویژگی های یک شی را به متغیرهایی با نام های مختلف اختصاص دهید:

|  |
| --- |
| const { johnDoe: { age: userAge, email: userEmail }} = user; |

تمرین2-13: دو انتساب را با یک Destructuring معادل جایگزین کنید. هنوز هم باید به متغیرهای lowToday و highToday مقادیر today.low و today.high را از شی LOCAL\_FORECASTاختصاص دهید.

|  |
| --- |
| const LOCAL\_FORECAST = {    yesterday: { low: 61, high: 75 },    today: { low: 64, high: 77 },    tomorrow: { low: 68, high: 80 }  };  const lowToday = LOCAL\_FORECAST.today.low;  const highToday = LOCAL\_FORECAST.today.high; |

## استفاده از Destructuring Assignment برای انتساب متغیرها از آرایه ها

ES6 باعث می تعقی ساختار آرایه ها به آسانی تعقیر ساختار اشیا باشد. یک تفاوت اساسی بین عملگرگسترش و تعقیر ساختار آرایه در این است که عملگر گسترش همه محتویات یک آرایه را در یک لیست جداشده با کاما باز می‌کند. در نتیجه ، شما نمی توانید عناصر مورد نظر خود رابرای انتساب به متغیرها بردارید یا انتخاب کنید. destructuring یک آرایه به ما امکان می دهد دقیقاً این کار را انجام دهیم:

|  |
| --- |
| const [a, b] = [1, 2, 3, 4, 5, 6];  console.log(a, b); // 1, 2 |

به متغیر a مقدار اول آرایه اختصاص داده می شود و به b مقدار دوم آرایه اختصاص می یابد. ما همچنین می‌توانیم با استفاده از کاما برای رسیدن به index مورد نظر ، به هر index در یک آرایه با destructuring دسترسی داشته باشیم:

|  |
| --- |
| const [a, b,,, c] = [1, 2, 3, 4, 5, 6];  console.log(a, b, c); // 1, 2, 5 |

تمرین2-14: از destructuring assignment برای جابجایی مقادیر a و b استفاده کنید تا a مقدار ذخیره شده در b را دریافت کند ، و b مقدار ذخیره شده در a را دریافت کند.

|  |
| --- |
| let a = 8, b = 6; |

## استفاده از Destructuring Assignment برای تنظیم مجدد عناصر با استفاده از پارمترRest

در بعضی شرایط که شامل Destructuring آرایه می شود ، ممکن است بخواهیم بقیه عناصر را در یک آرایه جداگانه جمع آوری کنیم. نتیجه مشابه با Array.prototype.slice() است ، همانطور که در زیر نشان داده شده است:

|  |
| --- |
| const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];  console.log(a, b); // 1, 2  console.log(arr); // [3, 4, 5, 7] |

متغیرهای a و b مقادیر اول و دوم را از آرایه می گیرند. پس از آن ، به دلیل وجود پارامتر rest ، arr بقیه مقادیر را به صورت آرایه می گیرد. عنصرrest فقط به عنوان آخرین متغیر لیست به درستی کار می کند. مانند گذشته ، شما نمی توانید از پارامتر استراحت برای گرفتن زیرآرایه ای استفاده کنید که از آخرین عنصر آرایه اصلی خارج می‌شود.

تمرین2-15: برای اجرای یک اثر Array.prototype.slice() از destructuring assignment با پارامتر rest استفاده کنید تا arr یک زیرآرایه از آرایه اصلی source باشد که دو عنصر اول از آنها حذف شده است.

|  |
| --- |
| const source = [1,2,3,4,5,6,7,8,9,10];  function removeFirstTwo(list) {    "use strict";    const arr = list; // این خط را تعقیر دهید    return arr;  }  const arr = removeFirstTwo(source); |

## استفاده از Destructuring Assignment برای ارسال یک شی به عنوان پارامترهای یک تابع

در بعضی موارد ، می توانید شیء را در یک آرگومان تابع تعقیر ساختار دهید. کد زیر را در نظر بگیرید:

|  |
| --- |
| const profileUpdate = (profileData) => {    const { name, age, nationality, location } = profileData;    // do something with these variables |

این به طور موثری شیء ارسال شده به تابع را تعقیر ساختار می‌دهد. این همچنین می تواند در محل انجام شود:

|  |
| --- |
| const profileUpdate = ({ name, age, nationality, location }) => {    /\* do something with these fields \*/  } |

این خطوط اضافی را از بین می برد و باعث می شود کد ما به نظر بی نظیر باشد. این مزیت اضافی عدم نیاز به دستکاری کل یک شیء در یک تابع است - فقط فیلدهای مورد نیاز در داخل تابع کپی می‌شوند.

تمرین2-16: از Destructuring Assignment در آرگومان تابع half استفاده کنید تا فقط max و min به داخل تابع ارسال شود.

|  |
| --- |
| const stats = {    max: 56.78,    standard\_deviation: 4.34,    median: 34.54,    mode: 23.87,    min: -0.75,    average: 35.85  };  const half = (stats) => (stats.max + stats.min) / 2.0; |

## ایجاد رشته ها با استفاده از Template Literal ها

template literal ویژگی جدید ES6 به معنای واقعی کلمه است. این نوع خاصی از رشته ها است که ایجاد رشته های پیچیده را آسان تر می کند.

template literal به شما امکان ایجاد رشته های چند خطی را می دهد و از ویژگی های درون یابی رشته ای برای ایجاد رشته ها استفاده می کنید. کد زیر را در نظر بگیرید:

|  |
| --- |
| const person = {    name: "Zodiac Hasbro",    age: 56  };  // Template literal with multi-line and string interpolation  const greeting = `Hello, my name is ${person.name}!  I am ${person.age} years old.`;  console.log(greeting); // prints  // Hello, my name is Zodiac Hasbro!  // I am 56 years old. |

اتفاقات زیادی در آنجا رخ داده است. در مرحله اول ، مثال برای پیچیدن رشته ها از backticks (`) و نه نقل قول ( ' یا ") استفاده می کند. ثانیا ، توجه کنید که رشته چند خطی است ، هم در کد و هم در خروجی. در وارد کردن \n در رشته ها صرفه جویی می شود. نحو ${variable} استفاده شده در بالا یک مکان نگهدارنده است. در اصل ، شما دیگر نیازی به استفاده از الصاق با عملگر + نخواهید داشت. برای افزودن متغیرها به رشته ها ، شما فقط متغیر را در یک template رشته می اندازید و آنرا با ${ } می پیچید.

به همین ترتیب ، شما می‌توانید عبارات دیگر را در متن رشته خود وارد کنید ، به عنوان مثال ${a + b}. این روش جدید ایجاد رشته ها ، انعطاف پذیری بیشتری را برای ایجاد رشته های قوی ایجاد می کند.

تمرین2-17: برای نمایش هر ورودی از آرایه failure شیء result از نحو template literalاستفاده کنید. هر ورودی باید درون یک تگ li با ویژگی کلاس text-warningو در قسمت resultDisplayArray لیست شود.

از یک متد تکرار شونده (هر نوع حلقه ای) برای به دست آوردن خروجی مورد نظر استفاده کنید (مانند شکل زیر).

[

  '<li class="text-warning">no-var</li>',

  '<li class="text-warning">var-on-top</li>',

  '<li class="text-warning">linebreak</li>'

]

|  |
| --- |
| const result = {    success: ["max-length", "no-amd", "prefer-arrow-functions"],    failure: ["no-var", "var-on-top", "linebreak"],    skipped: ["id-blacklist", "no-dup-keys"]  };  function makeList(arr) {    "use strict";    // Only change code below this line    const resultDisplayArray = null;    // Only change code above this line    return resultDisplayArray;  }  const resultDisplayArray = makeList(result.failure); |

## نوشتن تعریف های لیترال شیء مختصر با استفاده ازکوتاه کردن خاصیت شی

ES6 پشتیبانی خوبی را برای تعریف آسان لیترال هایی از اشیاء اضافه کرده است. کد زیر را در نظر بگیرید:

|  |
| --- |
| const getMousePosition = (x, y) => ({    x: x,    y: y  }); |

getMousePosition یک تابع ساده است که یک شیء حاوی دو خاصیت را برمی گرداند. ES6 نحوی را برای از بین بردن افزونگی نوشتن x: x فراهم می کند. شما می توانید به سادگی یک بار بنویسیدx ، و آن به x: x (یا چیزی معادل آن) تبدیل می شود. در اینجا برای استفاده از این نحو جدید همان تابع بالا بازنویسی شده است:

|  |
| --- |
| const getMousePosition = (x, y) => ({ x, y }); |

تمرین2-18: برای ایجاد و بازگرداندن یک شی با خواص name، age و gender از ویژگی کوتاه کردن خاصیت شی و لیترال های شی استفاده کنید.

|  |
| --- |
| const createPerson = (name, age, gender) => {    "use strict";    return {      name: name,      age: age,      gender: gender    };  }; |

## نوشتن توابع اعلانی مختصر را با ES6

هنگام تعریف توابع درون اشیاء در ES5 ، باید از کلید واژه function به شرح زیر استفاده کنیم:

|  |
| --- |
| const person = {    name: "Taylor",    sayHello: function() {      return `Hello! My name is ${this.name}.`;    }  }; |

با ES6 ، می توانید در هنگام تعریف توابع در اشیاء ، کلمه کلیدی function و کولن را حذف کنید. در اینجا نمونه ای از این نحو آورده شده است:

|  |
| --- |
| const person = {    name: "Taylor",    sayHello() {      return `Hello! My name is ${this.name}.`;    }  }; |

تمرین2-19: setGear را در داخل شی bicycle مورد استفاده قرار دهید تا از نحو کوتاه شده که در بالا گفته شد استفاده کنید.

|  |
| --- |
| const bicycle = {    gear: 2,    setGear: function(newGear) {      this.gear = newGear;    }  };  bicycle.setGear(3);  console.log(bicycle.gear); |

## استفاده از نحو class برای تعریف تابع سازنده

ES6 با استفاده از کلمه کلیدی class ، نحو جدیدی را برای ایجاد اشیاء فراهم می کند. لازم به ذکر است که نحو class فقط نحو است ، و یک اجرای کامل مبتنی بر کلاس از یک الگوی شی گرا بر خلاف زبانهایی مانند جاوا ، پایتون ، روبی و غیره نیست.

در ES5 ، معمولاً یک تابع سازنده را تعریف می کنیم و از کلمه کلیدی new برای ایجاد یک شیء استفاده می‌کنیم.

|  |
| --- |
| var SpaceShuttle = function(targetPlanet){    this.targetPlanet = targetPlanet;  }  var zeus = new SpaceShuttle('Jupiter'); |

نحو class به سادگی جایگزین ایجاد تابع سازنده می شود:

|  |
| --- |
| class SpaceShuttle {    constructor(targetPlanet) {      this.targetPlanet = targetPlanet;    }  }  const zeus = new SpaceShuttle('Jupiter'); |

لازم به ذکر است که کلمه کلیدی class یک تابع جدید را اعلان می‌کند ، که به آن یک سازنده اضافه می‌شود. این سازنده در هنگام فراخوانی new برای ایجاد یک شی جدید فراخوانی می شود.

به یاد داشته باشید که:

* از UpperCamelCase باید برای اسامی کلاس ES6 استفاده شود ، همانطور که در SpaceShuttle در بالا استفاده شده است.
* روش سازنده یك روش ویژه برای ایجاد و مقداردهی یك شیء است كه با یك كلاس ایجاد می شود. در مورد برنامه نویسی شی گرا در الگوریتم ها و ساختمان‌داده‌های جاوا اسکریپت اطلاعات بیشتری کسب خواهید کرد.

تمرین2-20: از کلمه کلیدی class استفاده کرده و یک سازنده برای ایجاد کلاس Vegetable بنویسید. کلاس Vegetable به شما امکان می دهد یک شیء vegetable با یک نام خاصیت ایجاد کنید که به سازنده ارسال می‌شود.

|  |
| --- |
| const carrot = new Vegetable('carrot');  console.log(carrot.name); // Should display 'carrot' |

## استفاده getter ها و setter ها برای کنترل دسترسی به یک شی

می توانید مقادیر یک شی را بدست آورید و مقدار یک ویژگی را در یک شی تنظیم کنید. این‌ها بصورت كلاسیك و getter (گیرنده)ها و setter (تنظیم کننده‌)ها نامیده می شوند.

توابع Getter به معنای ساده بازگشت مقدار متغیر خصوصی یک شی به کاربر است بدون اینکه کاربر مستقیماً به متغیر خصوصی دسترسی پیدا کند.

توابع Setter برای تغییر (تنظیم) مقدار متغیر خصوصی یک شیء بر اساس مقدار ارسال شده در تابع تنظیم کننده است. این تغییر ممکن است شامل محاسبات یا حتی بازنویسی مقدار قبلی به طور کامل باشد.

|  |
| --- |
| class Book {    constructor(author) {      this.\_author = author;    }    // getter    get writer() {      return this.\_author;    }    // setter    set writer(updatedAuthor) {      this.\_author = updatedAuthor;    }  }  const lol = new Book('anonymous');  console.log(lol.writer);  // anonymous  lol.writer = 'wut';  console.log(lol.writer);  // wut |

به نحو مورد استفاده برای فراخوانی گیرنده و تنظیم کننده توجه کنید. آنها حتی شبیه به توابع نیستند. گیرنده ها و تنظیم کننده ها مهم هستند زیرا جزئیات اجرای داخلی را پنهان می کنند. توجه: قرارداد است که قبل از نام متغیر خصوصی با uderscore (\_) قرار دهیم. با این حال ، خود عمل یک متغیر را خصوصی نمی کند.

تمرین2-21: برای ایجاد کلاس Thermostat از کلمه کلیدی class استفاده کنید. سازنده دمای فارنهایت را می پذیرد.

اکنون یک گیرنده و تنظیم کننده را در کلاس ایجاد کنید تا درجه حرارت در درجه سانتیگراد به دست آید. به یاد داشته باشید که C = 5/9 \* (F - 32) و F = C \* 9.0 / 5 + 32 که F در آن مقدار درجه حرارت در فارنهایت است و C مقدار همان درجه حرارت در Celsius است.

توجه: هنگام اجرای این کار ، دمای داخل class را در یک مقیاس ، فارنهایت یا سلسیوس پیگیری می‌کنید. این قدرت گیرنده و تنظیم کننده است. شما در حال ایجاد API برای یک کاربر دیگر هستید ، که می تواند صرف نظر از درنظر گرفتن هر کدام یک از آنها ، نتیجه صحیح را بدست آورد.

به عبارت دیگر ، شما در حال انتزاع جزئیات اجرا از کاربر هستید.

|  |
| --- |
| //فقط بالای این خط را تعقیر دهید  const thermos = new Thermostat(76); // Setting in Fahrenheit scale  let temp = thermos.temperature; // 24.44 in Celsius  thermos.temperature = 26;  temp = thermos.temperature; // 26 in Celsius  class Book {    constructor(author) {      this.\_author = author;    }    // getter    get writer() {      return this.\_author;    }    // setter    set writer(updatedAuthor) {      this.\_author = updatedAuthor;    }  }  const lol = new Book('anonymous');  console.log(lol.writer);  // anonymous  lol.writer = 'wut';  console.log(lol.writer);  // wut |

## ایجاد یک ماجول script

جاوا اسکریپت با یک نقش کوچک شروع به بازی کردن در یک وب سایت HTML که عمدتا متمایز است ، شد. امروزه ،آن بسیار بزرگ است و برخی وب سایت ها تقریباً کاملاً با JavaScript ساخته شده اند. به منظور اینکه جاوا اسکریپت بیشتر ماژولار ، تمیز و قابل دستکاری شود؛ ES6 راهی برای به اشتراک گذاری کد در میان فایل‌های JavaScript معرفی کرد.

این امر شامل استخراج قسمت هایی از یک فایل برای استفاده در یک یا چند فایل دیگر و وارد کردن قسمت‌های مورد نیاز در جایی که به آنها نیاز دارید ، می باشد. برای استفاده از این قابلیت ، شما باید یک اسکریپت را در سند HTML خود با یک نوع ماژول ایجاد کنید. در اینجا مثالی وجود دارد:

|  |
| --- |
| <script type="module" src="filename.js"></script> |

اسکریپتی که از این نوع module استفاده می کند ، هم اکنون می تواند از ویژگی های import و export که در چالش های آینده با آنها آشنا خواهید شد ، استفاده کند.

تمرین2-22: یک اسکریپت را به سند HTML از نوع module اضافه کنید و به آن منبع فایل index.js بدهید

|  |
| --- |
| <html>    <body>      <!-- Only change code below this line -->      <!-- Only change code above this line -->    </body>  </html> |

## استفاده از export برای به اشتراک گذاری یک بلوک کد

فایلی را با نام math\_functions.js تصور کنید که شامل چندین تابع مربوط به عملیات ریاضی است. یکی از آنها در یک متغیر ذخیره می شود ، add، که دو عدد می‌گیرد و مجموع آنها را برمی گرداند. شما می خواهید از این تابع در چندین فایل مختلف JavaScript استفاده کنید. برای به اشتراک گذاشتن آن با این فایل های دیگر ، ابتدا باید آن را export کنید.

|  |
| --- |
| export const add = (x, y) => {    return x + y;  } |

موارد فوق روشی متداول برای export یک تابع واحد است ، اما می توانید به موارد مشابه مانند این برسید:

|  |
| --- |
| const add = (x, y) => {    return x + y;  }  export { add }; |

هنگامی که یک متغیر یا تابع را export می کنید ، می توانید آن را در فایل دیگری import کنید و بدون نیاز به بازنویسی کد ، از آن استفاده کنید. با تکرار مثال اول، یا با قرار دادن همه آنها در یک دستور export از مثال دوم ، می توانید چندین چیز را export کنید،مانند این:

|  |
| --- |
| export { add, subtract }; |

تمرین2-23: دو تابع مبتنی بر رشته در ویرایشگر وجود دارد. هر دو را با استفاده از روش مورد نظر خود export کنید.

|  |
| --- |
| const uppercaseString = (string) => {    return string.toUpperCase();  }  const lowercaseString = (string) => {    return string.toLowerCase()  } |

## استفاده مجدد از کد جاوااسکریپت با استفاده از import

import به شما امکان می دهد قسمت هایی از یک فایل یا ماژول را برای بارگیری انتخاب کنید. در درس قبلی ، برای مثال add از فایل math\_functions.js ، export شد. در اینجا چگونگی import کردن آن برای استفاده در فایل دیگر آورده شده است:

|  |
| --- |
| import { add } from './math\_functions.js'; |

در اینجا import، add را از math\_functions.js پیدا خواهد کرد ، فقط آن تابع را برای استفادهimport کنید و بقیه را نادیده بگیرید. ./ به import می‌گوید که به دنبال فایل math\_functions.js در همان پوشه جاری باشد. مسیر نسبی فایل (./) و پسوند فایل (.js) هنگام استفاده از import در این روش لازم است.

شما می توانید بیش از یک مورد فایل را با اضافه کردن آنها در دستور import مانند این وارد کنید:

|  |
| --- |
| import { add, subtract } from './math\_functions.js'; |

تمرین2-24: دستور import مناسبی را اضافه کنید که به thdg فعلی امکان استفاده از توابع smallcaseString و smallcaseString را که در درس قبلی ecportکرده اید ، را بدهد. این توابع درفایل‌ای به نام string\_functions.js است که در همان شاخه موجود در کنار فایل فعلی قرار دارد.

|  |
| --- |
| uppercaseString("hello");  lowercaseString("WORLD!"); |

## استفاده از \* برای import کردن همه چیز از فایل

فرض کنید فایلی دارید و می خواهید تمام محتوای آن را در فایل فعلی import کنید. این را می توان با نحو import \* as انجام داد. در اینجا مثالی آورده شده است که محتوای فایلی به نام math\_functions.js در یک فایل در همان شاخهimport می شود:

|  |
| --- |
| import \* as myMathModule from "./math\_functions.js"; |

دستور import فوق یک شیء به نام myMathModule ایجاد می‌کند. این فقط یک نام متغیر است ، می توانید آن را هر چیزی بنامید. این شیء شامل همه export های موجود در math\_functions.js است ، بنابراین شما می‌توانید به توابع مانند هر خاصیت شی دیگر دسترسی داشته باشید. در اینجا نحوه استفاده از توابع add و subtract که import شده است آورده شده است:

|  |
| --- |
| myMathModule.add(2,3);  myMathModule.subtract(5,3); |

تمرین2-25: کد موجود در این پرونده به محتویات پرونده string\_functions.js نیاز دارد: آن در همان شاخه که پرونده فعلی هست، می باشد. از نحو import \* as استفاده کنید تا همه چیز را از فایل به شیء بنام stringFunctions ، import کنید.

|  |
| --- |
| uppercaseString("hello");  lowercaseString("WORLD!"); |

## ایجاد یک export fallback با export default

در درس export ، در مورد نحوی که به عنوان export نامگذاری شده یاد گرفتید. این به شما امکان می دهد چندین تابع و متغیر را برای استفاده در فایل های دیگر در دسترس داشته باشید.

نحو export دیگری که باید بدانید وجود دارد ، معروف به export default. به طور معمول شما از این نحو اگر فقط یک مقدار از یک فایل export شود ، استفاده خواهید کرد. همچنین برای ایجاد یک مقدار بازگشتی برای یک فایل یا ماژول استفاده می شود. در زیر مثال هایی با استفاده از export default وجود دارد:

|  |
| --- |
| // named function  export default function add(x, y) {    return x + y;  }  // anonymous function  export default function(x, y) {    return x + y;  } |

از آنجا که export default برای اعلان یک مقدار بازگشتی برای یک ماژول یا فایل استفاده می شود ، شما فقط می توانید یک مقدار export default در هر ماژول یا فایل داشته باشید. علاوه بر این ، شما نمی توانید export default را با var ، let یا const استفاده کنید.

تمرین2-26: تابع زیر باید مقدار بازگشتی برای یک ماژول باشد. لطفا کد لازم را برای این کار اضافه کنید.

|  |
| --- |
| function subtract(x, y) {    return x - y;  } |

## Import کردن یک default export

در آخرین چالش ، شما در مورد default export و کاربردهای آن اطلاعات کسب کردید. برای import یک default export ، باید از نحو import متفاوتی استفاده کنید. در مثال زیر ، add یک default export در فایل math\_functions.js است. در اینجا نحوه import کردن آن آمده است:

|  |
| --- |
| mport add from "./math\_functions.js"; |

نحو در یک محل یک کلید متفاوت است. مقدار import شده ، add، توسط پرانتزهای فرفری احاطه نشده است ({). add در اینجا به سادگی یک نام متغیر برایdefault export فایل math\_functions.js است. می‌توانید در هنگام import کردن پیش فرض از هر نام دیگری استفاده کنید.

تمرین2-27: در کد زیر ، default export را از پرونده math\_functions.js که در همان شاخه با این پرونده موجود است ، import کنید. به import نام subtract را بدهید.

|  |
| --- |
| // کد موردنظر را به بالای این خط اضافه کنید  subtract(7,4); |

## ایجاد یک promise جاوااسکریپت

یک promise در JavaScript دقیقاً همان چیزی است که به نظر می رسد - شما از آن استفاده می کنید تا یک یک وعده برای انجام کاری را ایجاد کنید ، معمولاً به صورت غیر همزمان. پس از اتمام کار ، شما یا به قول خود عمل می کنید یا در انجام آن ناکام هستید. promise یک تابع سازنده است ، بنابراین شما نیاز به استفاده از کلمه کلیدی new برای ایجاد آن داریدآن یکه تابع را به عنوان آرگومان ورودی، با دو پارامتر می‌گیرد – resolve و reject. این ها متدهایی هستند که برای تعیین نتیجه promise استفاده می شوند. نحو مانند این است:

|  |
| --- |
| const myPromise = new Promise((resolve, reject) => {  }); |

در جاوااسکریپت Promise نشان دهنده نتیجه یک عملیات ناهمزمان است . Promise رو میشه به عنوان یک نگهدارنده تصور کرد. این نگهدارنده در واقع یک object است که میتونیم callbacks رو بهش پیوست کنیم.

اغلب اوقاتی که با Promise کار میکنیم ، با Promiseهایی سروکار داریم که توسط توابع بازگشت داده شدن.با وجود این میتونیم  Promise رو در سازنده تعریف کنیم.

تمرین2-28: promise جدیدی بنام makeServerRequest ایجاد کنید. با استفاده از پارامترهای resolve و reject سازنده به تابع ارسال کنید.

## تکمیل یک promise با resolve و reject

یک promise سه حالت دارد: pending، fulfilled و rejected است. Promisای که شما در آخرین چالش ایجاد کردید ، برای همیشه در حالت pending(کامل نشده) گیر افتاده است زیرا راهی برای انجام promise اضافه نکردید. پارامترهای fulfilled و rejected به آرگومان promise برای این کار استفاده می‌شوند. درصورتی که می خواهید promise خود را به موفقیت برساند ، fulfilled استفاده می شود و هنگامی که می خواهید شکست بخورد ، از rejected استفاده می شود. این ها متد هایی هستند که یک آرگومان را می گیرند ، همانطور که در شکل زیر مشاهده می شود.

|  |
| --- |
| const myPromise = new Promise((resolve, reject) => {    if(condition here) {      resolve("Promise was fulfilled");    } else {      reject("Promise was rejected");    }  }); |

مثال بالا برای آرگومان این توابع از رشته ها استفاده می کند ، اما در واقع می تواند هر چیزی باشد. غالباً ممکن است این شیء باشد که شما از داده ها استفاده می کنید تا آنها در وب سایت تان یا جای دیگر قرار دهید. تمرین2-29: promise را با موفقیت و شکست روبرو کنید. اگر responseFromServer،true است ، برای تکمیل موفقیت آمیز promise متد resolve را فراخوانی کنید. به resolved مقدار We got the data را ارسال کنید. اگر responseFromServer، false است ، به جای آن از متد reject استفاده کنید و این رشته را به آن ارسال کنید: Data not received.

|  |
| --- |
| const makeServerRequest = new Promise((resolve, reject) => {    // responseFromServer represents a response from a server    let responseFromServer;    if(responseFromServer) {      // Change this line    } else {      // Change this line    }  }); |

## handle یک promise ، fulfilled با then

promise ها زمانی مفید هستند که فرایندی داشته باشید که مدت زمان ناشناخته ای را در کد شما به طول می‌انجامد (یعنی چیزی غیر همزمان) ، اغلب درخواست سرور. وقتی درخواست سرور می دهید زمان زیادی طول می کشد ، و پس از اتمام آن معمولاً می خواهید با پاسخ سرور کاری انجام دهید. این امر می تواند با استفاده از متد then حاصل شود. متد then بلافاصله پس از fulfilled، promise شما با resolve اجرا می شود. در اینجا مثالی وجود دارد:

|  |
| --- |
| myPromise.then(result => {    // do something with the result.  }); |

result از آرگومان داده شده به متد resolve بدست می آید.

تمرین2-30: متد then را به promise خود اضافه کنید. از result به عنوان پارامتر تابع بازگشتی استفاده کنیدو result را در کنسول نمایش دهید.

|  |
| --- |
| const makeServerRequest = new Promise((resolve, reject) => {    // responseFromServer is set to true to represent a successful response from a server    let responseFromServer = true;      if(responseFromServer) {      resolve("We got the data");    } else {      reject("Data not received");    }  }); |

## handle یک promise ، Rejected با catch

catch روشی است که در هنگام rejected، promise شما مورد استفاده قرار می گیرد. بلافاصله پس از فراخوانی متد rejected، promise اجرا می شود. نحو در اینجا آمده است:

|  |
| --- |
| myPromise.catch(error => {    // do something with the error.  }); |

error آرگومانی است که به متد reject ارسال می شود.

توجه: متد های then و catch در صورت انتخاب ، می توانند به تعریف های promise زنجیر شوند.

تمرین2-31: متد catch را به promise خود اضافه کنید. از error به عنوان پارامتر تابع بازگشتی استفاده کنید و error را در کنسول نمایش دهید.

|  |
| --- |
| const makeServerRequest = new Promise((resolve, reject) => {    // responseFromServer is set to true to represent a successful response from a server    let responseFromServer = true;      if(responseFromServer) {      resolve("We got the data");    } else {      reject("Data not received");    }  }); |

## 3. عبارات منظم

## آشنایی با چالش های عبارات منظم

عبارات منظم رشته های خاصی هستند که یک الگوی جستجو را نشان می دهند. آنها همچنین به عنوان "regex" یا "regexp" نیز شناخته می شوند ، آنها به برنامه نویسان کمک می کنند تا متن را مطابقت ، جستجو و جایگزین کنند. عبارات منظم می توانند به صورت رمزنگاری شده ظاهر شوند زیرا چند کاراکتر از اهمیت خاصی برخوردار هستند. هدف این است که نمادها و متن را با الگویی ترکیب کنید که مطابق آنچه می خواهید باشد ، اما فقط آنچه می خواهید باشد. این بخش کاراکترها ، چند کلید میانبر و کاربردهای متداول برای نوشتن عبارات منظم را پوشش می دهد.

## استفاده از متد Test

عبارات منظم در زبان های برنامه نویسی برای مطابقت با قسمت هایی از رشته ها استفاده می شود. شما الگوهایی را ایجاد می کنید تا به شما در انجام این تطبیق کمک کنند.

اگر می خواهید کلمه "the" را در رشته "The dog chased the cat" پیدا کنید ، می توانید از عبارت منظم زیر استفاده کنید: /the/. توجه کنید که علائم نقل قول در عبات منظم لازم نیست.

javaScript روش های مختلفی برای استفاده از regexes دارد. یک روش برای آزمایش یک regex استفاده از متد .test()است. متد . .test()، regex را می گیرد، آن را روی رشته (که داخل پرانتز قرار می گیرد) اعمال می‌کند ، و اگر الگوی شما چیزی پیدا کند یا نکند ، true یا false برمی گرداند.

|  |
| --- |
| let testStr = "freeCodeCamp";  let testRegex = /Code/;  testRegex.test(testStr);  // Returns true |

تمرین3-1: regex، myRegex را با استفاده از متد .test()روی رشته myString اعمال کنید.

|  |
| --- |
| let myString = "Hello, World!";  let myRegex = /Hello/;  let result = myRegex; // Change this line |

## مطابقت داشتن با رشته های لیترال

در آخرین چالش ، شما با استفاده از عبارت منظم /Hello/ کلمه "Hello"را جستجو کردید. آن regex در جستجوی لیترال "Hello"از رشته بود. مثال دیگری در جستجو برای لیترال "Kevin" از رشته است:

|  |
| --- |
| let testStr = "Hello, my name is Kevin.";  let testRegex = /Kevin/;  testRegex.test(testStr);  // Returns true |

هر شکل دیگری از "Kevin"مطابقت نخواهد داشت. به عنوان مثال regex، /Kevin/با "kevin" یا "KEVIN" مطابقت نخواهد داشت.

|  |
| --- |
| let wrongRegex = /kevin/;  wrongRegex.test(testStr);  // Returns false |

یکی از چالش های آینده نشان می دهد که چگونه می توان آن اشکال دیگر را مطابقت داد.

تمرین3-2: regex، waldoRegexرا تکمیل کنید تا "Waldo" را در رشته waldoIsHiding با یک مطابقت لیترال پیدا کنید.

|  |
| --- |
| let waldoIsHiding = "Somewhere Waldo is hiding in this text. ";  let waldoRegex = /search/; // این خط را تعقیر دهید  let result = waldoRegex.test(waldoIsHiding); |

## مطابقت تعداد مختلف با یک رشته واقعی

با استفاده از regex هایی مانند /coding/، می توانید الگوی "coding"را در رشته دیگری جستجو کنید. این برای جستجوی تک رشته ها قدرتمند است ، اما فقط به یک الگو محدود می شود. می توانید با استفاده از عملگر alternation یا OR گزینه های مختلفی را جستجو کنید: |.

این عملگر همچنین با الگوهای قبل یا بعد از آن مطابقت دارد. به عنوان مثال ، اگر می خواستید "yes"یا "no"را مطابقت دهید ، regex مورد نظر شما این است /yes|no/.

همچنین می توانید بیش از دو الگو را جستجو کنید. شما می توانید این کار را با اضافه کردن الگوهای بیشتر با عملگر OR که آنها را جدا می کند ، مانند /yes|no|maybe/انجام دهید.

تمرین3-3: regex، petRegex را تکمیل کنید تا با حیوانات خانگی "dog"، "cat"، "bird"یا "fish" مطابقت داشته باشد.

|  |
| --- |
| let petString = "James has a pet cat.";  let petRegex = /change/; // این خط را تعقیر دهید  let result = petRegex.test(petString); |

## نادیده گرفتن Case درحین مطابقت

تاکنون ، شما برای انجام مطابقت لیترال منطبق رشته ها به regex ها نگاه کرده اید. اما گاهی اوقات ، ممکن است شما بخواهید تفاوت های Case را نیز مطابقت دهید.

Case (یا گاهی اوقات حروف) تفاوت بین حروف بزرگ و حروف کوچک است. نمونه هایی از حرف بزرگ "A" ، "B" و "C" است. نمونه هایی از حروف کوچک "a" ، "b" و "c" هستند.

می توانید هر دو case هارا با استفاده از آنچه پرچم نامیده می شود مطابقت دهید. پرچم های دیگری نیز وجود دارند اما در اینجا شما روی پرچمی تمرکز خواهید کرد که case را نادیده می گیرد - پرچم . i

می توانید با افزودن آن به regex از آن استفاده کنید. نمونه ای از استفاده از این پرچم /ignorecase/i است. این regex می تواند با رشته های "ignorecase" ، "igNoreCase" و "IgnoreCase" مطابقت داشته باشد.

تمرین3-4: regex ، fccRegex را برای مطابقت با "freeCodeCamp" بنویسید، case آن مهم نیست. regex شما نباید با هر اختصار یا تغییراتی با فاصله ها مطابقت داشته باشد.

|  |
| --- |
| let myString = "freeCodeCamp";  let fccRegex = /change/; // این خط را تعقیر دهید  let result = fccRegex.test(myString); |

## مطابقت Exrract

تاکنون فقط بررسی کرده اید که آیا الگویی در یک رشته وجود دارد یا نه. همچنین می توانید مطابق واقعی خود را با متد .match() پیدا کنید. برای استفاده از متد .match() را روی یک رشته بکار ببرید و regex را در داخل پرانتز قرار دهید.

در اینجا مثالی آورده شده است:

|  |
| --- |
| "Hello, World!".match(/Hello/);  // Returns ["Hello"]  let ourStr = "Regular expressions";  let ourRegex = /expressions/;  ourStr.match(ourRegex);  // Returns ["expressions"] |

توجه داشته باشید که نحو .match"نقطه مقابل" متد .test است که شما تاکنون استفاده کرده اید:

|  |
| --- |
| 'string'.match(/regex/);  /regex/.test('string'); |

تمرین3-5: برای extract کلمه coding ، از متد .match() استفاده کنید.

|  |
| --- |
| let extractStr = "Extract the word 'coding' from this string.";  let codingRegex = /change/; // این خط را تعقیر دهید  let result = extractStr; // این خط را تعقیر دهید |

## یافتن بیشتر از اولین مطابقت

تاکنون فقط یک بار توانستید الگویی را extract یا جستجو کنید.

|  |
| --- |
| let testStr = "Repeat, Repeat, Repeat";  let ourRegex = /Repeat/;  testStr.match(ourRegex);  // Returns ["Repeat"] |

برای جستجو یا extract یک الگو بیش از یک بار ، می توانید از پرچم g استفاده کنید.

|  |
| --- |
| let repeatRegex = /Repeat/g;  testStr.match(repeatRegex);  // Returns ["Repeat", "Repeat", "Repeat"] |

تمرین3-6: با استفاده regex، starRegex هر دو کلمه "Twinkle" را از رشته twinkleStar پیدا کرده و استخراج کنید.

نکنه: می توانید چندین پرچم در regex مانند /search/gi داشته باشید

|  |
| --- |
| let twinkleStar = "Twinkle, twinkle, little star";  let starRegex = /change/; // این خط را تعقیر دهید let result = twinkleStar; // این خط را تعقیر دهید |

## هر چیزی را با period، Wildcard مطابقت دهید

بعضی اوقات شما کاراکترهای دقیق در الگوهای خود را نمی خواهید (یا نیازشان ندارید). فکر کردن به همه کلماتی که مطابقت دارند ، مثلاً یک غلط املایی طول می کشد. خوشبختانه ، می توانید با استفاده از کاراکتر Wildcard در وقت خود صرفه جویی کنید: .

کاراکتر Wildcard . با هر یک از کاراکترها مطابقت دارد. wildcard همچنین dot و period نامیده می شود. شما می توانید از کاراکتر های wildcard درست مانند هر کاراکتر دیگری در regex استفاده کنید. به عنوان مثال ، اگر می خواستید "hug", "huh", "hut" و "hum" را مطابقت دهید ، می توانید از regex، /hu./ برای مطابقت با هر چهار کلمه استفاده کنید.

|  |
| --- |
| let humStr = "I'll hum a song";  let hugStr = "Bear hug";  let huRegex = /hu./;  huRegex.test(humStr); // Returns true  huRegex.test(hugStr); // Returns true |

تمرین3-7: regex ، unRegex را کامل کنید تا با رشته های "run", "sun", "fun", "pun", "nun", و "bun" مطابقت داشته باشد. regex شما باید از کاراکتر wildcard استفاده کند.

|  |
| --- |
| let exampleStr = "Let's have fun with regular expressions!";  let unRegex = /change/; // این خط را تعقیر دهید  let result = unRegex.test(exampleStr); |

## مطابقت با کاراکترهای مجزا با امکانات متعدد

شما آموخته اید که چگونه با الگوهای لیترال (/literal/) و کاراکتر wildcard (/./) مطابقت دهید. این موارد افراط و عبارات منظم هستند ، جایی که یکی مطابقت دقیقی پیدا می کند و دیگری با همه چیز مطابقت دارد. گزینه هایی وجود دارد که تعادل بین دو افراط وجود دارد.

با کلاس های کاراکتر می توانید یک الگوی لیترال را جستجو کنید. کلاس های کاراکتر به شما امکان می دهد با قرار دادن آنها در داخل براکت های مربع ([و]) گروهی از کاراکترهای مورد نظر خود را برای آنها تعریف کنید. به عنوان مثال ، شما می خواهید با "bag" ، "big" و "bug" مطابقت داشته باشند اما "bog" نباشد. برای این کار می توانید regex، /b[aiu]g/ را ایجاد کنید. [aiu] کلاس کاراکترهایی است که فقط با کاراکتر های "a", "i" یا "u" مطابقت خواهد داشت.

|  |
| --- |
| let bigStr = "big";  let bagStr = "bag";  let bugStr = "bug";  let bogStr = "bog";  let bgRegex = /b[aiu]g/;  bigStr.match(bgRegex); // Returns ["big"]  bagStr.match(bgRegex); // Returns ["bag"]  bugStr.match(bgRegex); // Returns ["bug"]  bogStr.match(bgRegex); // Returns null |

تمرین3-8: از یک کلاس کاراکتر با حروف صدادار (a, e, i, o, u) در regex، vowelRegex خود استفاده کنید تا همه مصوت ها را در عبارت quoteSample پیدا کنید.

توجه داشته باشید: مطمئن باشید که هر دو مصوت های بزرگ و کوچک را مطابقت دهید.

|  |
| --- |
| let quoteSample = "Beware of bugs in the above code; I have only proved it correct, not tried it.";  let vowelRegex = /change/; // این خط را تعقیر دهید  let result = vowelRegex; // این خط را تعقیر دهید |

## مطابقت با حروف الفبا

شما دیدید که چگونه می توانید از مجموعه کاراکترها برای مشخص کردن گروهی از کاراکترها برای مطابقت استفاده کنید ، اما در هنگام مطابقت دادن طیف وسیعی از کاراکترها (مثلاً هر حرف الفبایی) تایپ کردن زیاد است. خوشبختانه یک ویژگی داخلی نیز وجود دارد که این امر را کوتاه و ساده می کند.

در داخل یک مجموعه کاراکتر ، می توانید طیف وسیعی از کاراکترها را برای مطابقت با استفاده از یک کاراکتر hyphen تعریف کنید: -.

به عنوان مثال ، برای مطابقت با حروف کوچک a تا e از [a-e] استفاده می کنید.

|  |
| --- |
| let catStr = "cat";  let batStr = "bat";  let matStr = "mat";  let bgRegex = /[a-e]at/;  catStr.match(bgRegex); // Returns ["cat"]  batStr.match(bgRegex); // Returns ["bat"]  matStr.match(bgRegex); // Returns null |

تمرین3-9: همه حروف داخل رشته quoteSampleرا مطابقت دهید.

توجه داشته باشید: حتماً با هر دو حروف بزرگ و کوچک مطابقت داشته باشد.

|  |
| --- |
| let quoteSample = "The quick brown fox jumps over the lazy dog.";  let alphabetRegex = /change/; // این خط را تعقیر دهید  let result = alphabetRegex; // این خط را تعقیر دهید |

## مطابقت با اعداد و حروف الفبا

استفاده از hyphen (-) برای مطابقت با طیف وسیعی از کاراکترها فقط به حروف محدود نمی شود. همچنین برای با طیف وسیعی از اعداد هم کار می کند. به عنوان مثال ، /[0-5]/ با هر عددی بین 0 تا 5 از جمله 0 و 5 مطابقت دارد. همچنین می توان طیف وسیعی از حروف و اعداد را در یک مجموعه کاراکتر واحد ترکیب کرد.

|  |
| --- |
| let jennyStr = "Jenny8675309";  let myRegex = /[a-z0-9]/ig;  // matches all letters and numbers in jennyStr  jennyStr.match(myRegex); |

تمرین3-10: یک regex منفرد ایجاد کنید که متناسب با طیف وسیعی از حروف بین h وs ، و طیف وسیعی از اعداد بین 2 تا 6 باشد. بخاطر داشته باشید که پرچم های مناسب را در regex قرار دهید.

|  |
| --- |
| let quoteSample = "Blueberry 3.141592653s are delicious.";  let myRegex = /change/; // این خط را تعقیر دهید  let result = myRegex; // این خط را تعقیر دهید |

## مطابقت با کاراکترهای منفرد مشخص نشده

تاکنون مجموعه ای از کاراکترها را ایجاد کرده اید که می خواهید آنها را مطابقت دهید اما می توانید مجموعه کاراکترهایی را نیز ایجاد کنید که نمی خواهید آنها را مطابقت دهید. به این نوع مجموعه های کاراکتر مجموعه های کاراکترهای منفی گفته می شود.

برای ایجاد یک مجموعه کاراکتر منفی ، بعد از باز شدن براکت و قبل از کاراکترهایی که نمی خواهید مطابقت داشته باشند ، یک کاراکتر caret (^) را قرار دهید.

به عنوان مثال ، /[^aeiou]/gi با تمام کاراکتر هایی که صدادار نیستند مطابقت دارد. توجه داشته باشید که کاراکترهایی مانند ، ., !, [, @, / و فضای سفید مطابقت دارند - مجموعه کاراکترهای صدادار منفی فقط کاراکتر های صدادار را پیدا می کنند.

تمرین3-11: یک regex منفرد ایجاد کنید که با همه کاراکترهایی که یک عدد یا حرف صدادار نیستند مطابقت داشته باشد. به یاد داشته باشید که پرچمهای مناسب را در regex درج کنید.

|  |
| --- |
| let quoteSample = "3 blind mice.";  let myRegex = /change/; // این خط را تعقیر دهید  let result = myRegex; // این خط را تعقیر دهید |

## مطابقت دادن با کاراکترهایی که یک یا چند بار تکرار می شوند

بعضی اوقات ، شما باید یک کاراکتر (یا گروهی از کاراکتر ها) را که یک یا چند بار در یک ردیف ظاهر می شود ، مطابقت دهید. این بدان معنی است که حداقل یک بار اتفاق می افتد و ممکن است تکرار شود.

می توانید از کاراکتر + استفاده کنید تا صحت این مورد را ببینید. به یاد داشته باشید ، کاراکتر یا الگوی باید به صورت متوالی ظاهر شده باشد. یعنی کاراکتر باید یکی پس از دیگری تکرار شود.

به عنوان مثال ، /a+/g یک مطابقت را در "abc" پیدا می کند و ["a"] را برمی گرداند. به دلیل + ، همچنین یک مطابقت منفرد را در "aabc" پیدا می کند و ["aa"] بازمی‌گرداند.

اگر به جای آن رشته "abab" را چک می کرد ، دو مطابقت پیدا می کرد و ["a", "a"] بازمی‌گرداند زیرا کاراکتر ها در یک ردیف نیستند - یک b بین آنها وجود دارد. سرانجام ، از آنجا که "a" در رشته "bcd" وجود ندارد ، مطابقتی پیدا نمی کند.

تمرین3-12: شما میخواهید وقتی حرف s در "Mississippi"یکبار یا بیشتر تکرار می شود مطابقت هارا پیدا کنید. regex ای بنویسید که از علامت + استفاده می کند.

|  |
| --- |
| let difficultSpelling = "Mississippi";  let myRegex = /change/; // این خط را تعقیر دهید  let result = difficultSpelling.match(myRegex); |

## کاراکترهایی را که صفر یا چندبارظاهرمی‌شوند را مطابقت دهید

در آخرین چالش از علامت + plus برای جستجوی کاراکترهایی استفاده شد که یک یا چند بار ظاهر می‌شدند. همچنین گزینه ای وجود دارد برای مطابقت کاراکترهایی استفاده می‌شود که صفر یا چند بار ظاهر می‌شوند.

کاراکتر برای انجام این کار ضرب یا ستاره است: \*.

|  |
| --- |
| let soccerWord = "gooooooooal!";  let gPhrase = "gut feeling";  let oPhrase = "over the moon";  let goRegex = /go\*/;  soccerWord.match(goRegex); // Returns ["goooooooo"]  gPhrase.match(goRegex); // Returns ["g"]  oPhrase.match(goRegex); // Returns null |

تمرین3-13: برای این چالش ، chewieQuote با "Aaaaaaaaaaaaaaaarrrgh!" مقداردهی شده است. یک regex، chewieRegex ایجاد کنید که از کاراکتر \* برای مطابقت با کاراکتر "A" بزرگ استفاده کند و بلافاصله با کاراکترهای "a" کوچک صفر یا بیشتر در chewieQuote دنبال شود regex. شما نیازی به پرچم یا کلاس کاراکتر ندارد و نباید با هیچ یک از نقل قول های دیگر مطابقت داشته باشد.

|  |
| --- |
| let chewieRegex = /change/;  let result = chewieQuote.match(chewieRegex); |

## کاراکترها را با تطبیق تنبل پیدا کنید

در عبارات منظم ، یک مطابقت حریصانه طولانی ترین قسمت ممکن از یک رشته را پیدا می کند که متناسب با الگوی regex است و آن را به عنوان یک مطابقت باز می گرداند. جایگزین یک مطابقت تنبل نامیده می شود ، که کوچکترین قسمت ممکن از رشته را پیدا می کند که الگوی regex را برآورده کند.

می توانید regex، /t[a-z]\*i/ را روی رشته "titanic" اعمال کنید. این regex اساساً الگویی است که از t شروع می شود ، با i به پایان می رسد و در بین آنها حروفی وجود دارد.

عبارات منظم به طور پیش فرض حریصانه هستند ، بنابراین مطابقت ["titani"] را بازخواهد گرداند. آن بزرگترین رشته فرعی ممکن را که می تواند متناسب با الگو باشد را پیدا می‌کند.

با این حال ، شما می توانید از کاراکتر ؟ برای تغییر آن به مطابقت تنبل استفاده کنید. "titanic" در برابر regex تنظیم شده /t[a-z]\*?i/ ،["ti"] باز می گرداند.

توجه داشته باشید: از تجزیه HTML با عبارات منظم باید اجتناب شود ، اما الگوی تطبیق یک رشته HTML با عبارات منظم کاملاً خوب است.

تمرین3-14: عبارت منظم /<.\*>/ را برای بازگرداندن تگ <h1> ، HTML ونه متن

"<h1>Winter is coming</h1>" اصلاح کنید. به یاد داشته باشید که wildcard . در عبارت منظم با هر کاراکتری مطابقت دارد.

|  |
| --- |
| let text = "<h1>Winter is coming</h1>";  let myRegex = /<.\*>/; // این خط را تعقیر دهید  let result = text.match(myRegex); |

تمرین 3-15: یک یا چند جنایتکار را در یک شکار پیدا کنید

زمان مکث و تست مهارت های جدید نوشتن regex تان است. گروهی از جنایتکاران از زندان فرار کرده اند ، اما شما نمی دانید که چند نفرند. با این حال ، شما می دانید وقتی که آن ها در اطراف مردم هستند ، در کنار یکدیگر می مانند. شما مسئول پیدا کردن همه مجرمان به طور هم زمان هستید.

در اینجا مثالی برای بررسی نحوه انجام این کار آورده شده است:

عبارت منظم /z+/ با حرف z اگر یک یا چند بار در یک خط ظاهر شود مطابقت دارد. این مطابقت در همه رشته های زیر پیدا می شود:

|  |
| --- |
| "z"  "zzzzzz"  "ABCzzzz"  "zzzzABC"  "abczzzzzzzzzzzzzzzzzzzzzabc" |

اما در رشته های زیر مطابقت ندارد زیرا هیچ حرف z ای وجود ندارد:

|  |
| --- |
| ""  "ABC"  "abcabc" |

یک عبارت منظم حریصانه بنویسید که یک یا چند جنایتکار را در گروهی از افراد دیگر پیدا می کند. یک جنایتکار با حرف بزرگ C نشان داده می شود.

|  |
| --- |
| let reCriminals = /./; // این خط را تعقیر دهید |

## مطابقت الگوهای ابتدای رشته

چالش های قبلی نشان می داد که می توان از عبارات منظم برای جستجوی مطابقت یک عدد استفاده کرد. آنها همچنین برای جستجوی الگوها در موقعیت های خاص رشته ها استفاده می شوند.

در یک چالش قبلی ، شما از کاراکتر caret (^) در داخل یک کاراکتر برای ایجاد یک کاراکتر نفی شده به شکل [^thingsThatWillNotBeMatched] استفاده کرده اید. خارج از یک مجموعه کاراکتر ، از caret برای جستجوی الگوهای در ابتدای رشته ها استفاده می شود.

|  |
| --- |
| let firstString = "Ricky is first and can be found.";  let firstRegex = /^Ricky/;  firstRegex.test(firstString);  // Returns true  let notFirst = "You can't find Ricky now.";  firstRegex.test(notFirst);  // Returns false |

تمرین3-16: برای پیدا کردن "Cal" در ابتدای رشته rickyAndCal ، از کاراکتر caret در یک regex استفاده کنید.

|  |
| --- |
| let rickyAndCal = "Cal and Ricky both like racing.";  let calRegex = /change/; // Change this line  let result = calRegex.test(rickyAndCal); |

## مطابقت الگوهای انتهای رشته

در آخرین چالش ، شما یاد گرفتید که از کاراکتر caret برای جستجوی الگوها در ابتدای رشته ها استفاده کنید. همچنین راهی برای جستجوی الگوها در انتهای رشته ها وجود دارد. می توانید انتهای رشته ها را با استفاده از علامت دلار $ در انتهای regex جستجو کنید.

|  |
| --- |
| let theEnding = "This is a never ending story";  let storyRegex = /story$/;  storyRegex.test(theEnding);  // Returns true  let noEnding = "Sometimes a story will have to end";  storyRegex.test(noEnding);  // Returns false |

تمرین3-17: از کاراکتر لنگر ($) برای مطابقت با رشته "caboose" در انتهای رشته cabooseاستفاده کنید.

|  |
| --- |
| let caboose = "The last car on a train is the caboose";  let lastRegex = /change/; // Change this line  let result = lastRegex.test(caboose); |

## همه حروف و اعداد را مطابقت دهید

با استفاده از کلاس های کاراکتر ، شما قادر به جستجوی تمام حروف الفبا با [a-z] هستید. این نوع کلاس کاراکتر به اندازه کافی متداول است که میانبر برای آن وجود دارد ، اگرچه شامل چند کاراکتر اضافی نیز می‌شود.

نزدیکترین کلاس کاراکتر جاوا اسکریپت برای مطابقت با الفبا \wاست. این میانبر برابر است با

[A-Za-z0-9\_]. این کلاس کاراکتر با حروف بزرگ و کوچک به اضافه اعداد مطابقت دارد. توجه داشته باشید ، این کلاس کاراکتر شامل کاراکتر underscore (\_) نیز می باشد.

|  |
| --- |
| let longHand = /[A-Za-z0-9\_]+/;  let shortHand = /\w+/;  let numbers = "42";  let varNames = "important\_var";  longHand.test(numbers); // Returns true  shortHand.test(numbers); // Returns true  longHand.test(varNames); // Returns true  shortHand.test(varNames); // Returns true |

این کلاس های کاراکتر میانبر همچنین به عنوان کلاس های کاراکتر shorthand شناخته می شوند.

تمرین3-18: از کلاس کاراکتر shorthand ، \w استفاده کنید تا تعداد حروف الفبا را در نقل قول ها و رشته های مختلف بشمارید.

|  |
| --- |
| let quoteSample = "The five boxing wizards jump quickly.";  let alphabetRegexV2 = /change/; // Change this line  let result = quoteSample.match(alphabetRegexV2).length; |

## مطابقت همه موارد با حروف و اعداد

آموختید که می توانید از یک میانبر برای مطابقت الفبایی [A-Za-z0-9\_] از \w استفاده کنید. یک الگوی طبیعی که ممکن است بخواهید جستجو کنید ، متضاد الفبایی است.

می توانید برعکس \w را با \W جستجو کنید. توجه داشته باشید ، الگوی مخالف از یک حرف بزرگ استفاده می‌کند. این میانبر همان است [^A-Za-z0-9\_].

|  |
| --- |
| let shortHand = /\W/;  let numbers = "42%";  let sentence = "Coding!";  numbers.match(shortHand); // Returns ["%"]  sentence.match(shortHand); // Returns ["!"] |

تمرین3-19: از کلاس کاراکتر shorthand ، \W استفاده کنید تا تعداد کاراکترهای غیر الفبایی را در نقل قول ها و رشته های مختلف محاسبه کنید.

|  |
| --- |
| let quoteSample = "The five boxing wizards jump quickly.";  let alphabetRegexV2 = /change/; // Change this line  let result = quoteSample.match(alphabetRegexV2).length; |

## همه شماره ها را مطابقت دهید

شما میانبرها را برای الگوهای رشته های معمولی مانند alphanumeric ها یاد گرفتید. الگوی رایج دیگر فقط در جستجوی ارقام یا اعداد است. میانبر برای جستجوی کاراکترهای رقمی \d است ، با حرف کوچک d. این برابر با کلاس کاراکتر [0-9] است ، که به دنبال یک کاراکتر واحد از هر عددی بین صفر تا نه است.

تمرین3-20: از کلاس کاراکتر shorthand ، \d استفاده کنید تا تعداد رقم های موجود در movieName را شمارش کنید. شماره های نوشته شده "six" به جای 6) حساب نمی شوند.

|  |
| --- |
| let movieName = "2001: A Space Odyssey";  let numRegex = /change/; // این خط را تعقیر دهید  let result = movieName.match(numRegex).length; |

## همه غیر شماره ها را مطابقت دهید

آخرین چالش نحوه جستجوی ارقام با استفاده از میانبر \d با حرف کوچک d نشان داد. همچنین می توانید از میانبر مشابه که در عوض از D بزرگ استفاده می کند ، برای جستجوی کاراکتر های غیر رقمی استفاده کنید.

تمرین3-21: از کلاس کاراکتر shorthand ، \D استفاده کنید تا تعداد کاراکترهای غیر رقمی در movieNameرا بشمارید.

|  |
| --- |
| let movieName = "2001: A Space Odyssey";  let numRegex = /change/; // این خط را تعقیر دهید  let result = movieName.match(numRegex).length; |

## تمرین3-22: نام های کاربری ممکن را محدود کنید

نام های کاربری در همه جای اینترنت استفاده می شود. آنها همان چیزی هستند که هویت منحصر به فردی در سایت های مورد علاقه‌شان به کاربران می دهند. شما باید تمام نام‌های کاربری را در یک بانک اطلاعاتی بررسی کنید. در اینجا چند قانون ساده وجود دارد که کاربران هنگام ایجاد نام کاربری خود باید از آن‌ها پیروی کنند.

1. نام‌های کاربری فقط می توانند از کاراکترهای alpha-numeric استفاده کنند.
2. تنها شماره های موجود در نام کاربری باید در انتها باشد. در پایان صفر یا بیشتر از آنها وجود دارد. نام کاربری با شماره شروع نمی شود.
3. حروف نام کاربری می توانند کوچک و بزرگ باشند.
4. نام کاربری باید حداقل دو کاراکتر داشته باشد. یک نام کاربری دو کاراکتر فقط می تواند از حروف الفبا به عنوان کاراکتر استفاده کند.

برای بررسی محدودیت های ذکر شده در بالا ، regex ، userCheck را تغییر دهید.

|  |
| --- |
| let username = "JackOfAllTrades";  let userCheck = /change/; // این خط را تعقیر دهید  let result = userCheck.test(username); |

## مطابقت با فضای سفید

چالش های موجود تاکنون حروف مطابق الفبا و اعداد را پوشش داده است. همچنین می توانید فضای سفید یا فاصله بین حروف را مطابقت دهید. می توانید با استفاده از\s ، که یک حرف کوچک است ، فضای سفید را جستجو کنید. این الگو نه تنها با فضای سفید مطابقت دارد ، بلکه همچنین با carriage return ، tab ، form feed و کاراکترهایnew line مطابقت دارد. می توانید به آن به عنوان کلاس کاراکتر مشابه[\r\t\f\n\v] نگاه کنید.

|  |
| --- |
| let whiteSpace = "Whitespace. Whitespace everywhere!"  let spaceRegex = /\s/g;  whiteSpace.match(spaceRegex);  // Returns [" ", " "] |

تمرین3-23: regex ، countWhiteSpace را تغییر دهید تا به دنبال چندین کاراکتر فضای سفید در یک رشته باشید.

|  |
| --- |
| let sample = "Whitespace is important in separating words";  let countWhiteSpace = /change/; // Change this line  let result = sample.match(countWhiteSpace); |

## مطابقت با کاراکتر های غیر فضای سفید

شما در مورد جستجوی فضای سفید با استفاده از \s ، با حروف کوچک آموخته اید. همچنین می توانید همه چیز را به جز فضای سفید جستجو کنید. با استفاده از \S ، که یک حرف بزرگ است ، فضای غیر سفید را جستجو کنید. این الگو با whitespace، carriage return، tab ، form feed و کاراکتر new line مطابقت نخواهد داشت. می توانید تصور کنید که شبیه به کلاس کاراکتر [^ \r\t\f\n\v]است.

|  |
| --- |
| let whiteSpace = "Whitespace. Whitespace everywhere!"  let nonSpaceRegex = /\S/g;  whiteSpace.match(nonSpaceRegex).length; // Returns 32 |

تمرین3-24: regex ، countNonWhiteSpace را تغییر دهید تا به دنبال چندین کاراکتر غیر فضای سفید در یک رشته باشید.

|  |
| --- |
| let sample = "Whitespace is important in separating words";  let countWhiteSpace = /change/; // Change this line  let result = sample.match(countWhiteSpace); |

## تعداد بالا و پایین را برای مطابقت مشخص منید

به یاد بیاورید که برای جستجوی یک یا چند کاراکتر از + و برای جستجوی صفر یا چند کاراکتر از علامت ستاره \* استفاده کردید. اینها راحت هستند اما گاهی اوقات شما می خواهید طیف خاصی از الگوها را مطابقت دهید.

شما می توانید تعدادبزرگ و کوچک الگوها را با مشخصه کمیت مشخص کنید. مشخصه کمیت با براکت های فرفری ({و}) استفاده می شوند. دو عدد را بین {} قرار دهید - دو عدد را بین {} قرار دهید - برای تعداد حروف کوچک و بزرگ الگوها.

به عنوان مثال ، برای مطابقت با حرف aکه بین 3 تا 5 بار در رشته "ah" ظاهر می شود ، regex شما برابر خواهد بود باa{3,5}h/.

|  |
| --- |
| let A4 = "aaaah";  let A2 = "aah";  let multipleA = /a{3,5}h/;  multipleA.test(A4); // Returns true  multipleA.test(A2); // Returns false |

تمرین3-25: regex ، ohRegex را تغییر دهید تا مطابق با کلمه "Oh no" فقط در صورتی که دارای 3 تا 6 حرف h باشد ، مطابقت داشته باشد.

|  |
| --- |
| let ohStr = "Ohhh no";  let ohRegex = /change/; // Change this line  let result = ohRegex.test(ohStr); |

## فقط تعداد پایین را برای مطابقت مشخص کنید

با استفاده از {} می توانید تعداد پایین و بالای الگوها را با مشخصه های کمیت مشخص کنید. بعضی اوقات فقط می خواهید تعداد پایین الگوها را بدون حد بالایی مشخص کنید. برای مشخص کردن تعداد کمتر الگوها، اولین عدد را با کامابه دنبال آن بنویسید.

به عنوان مثال ، برای مطابقت با رشته "hah" با حرف a حداقل 3 بار ظاهر می شود ، regex شما برابر با /ha{3,}h/ خواهد بود.

|  |
| --- |
| let A4 = "haaaah";  let A2 = "haah";  let A100 = "h" + "a".repeat(100) + "h";  let multipleA = /ha{3,}h/;  multipleA.test(A4); // Returns true  multipleA.test(A2); // Returns false  multipleA.test(A100); // Returns true |

تمرین3-26: regex ، haRegex را تغییر دهید تا با کلمه "Hazzah" فقط در صورتی که دارای چهار یا بیشتر حرف z باشد مطابقت داشته باشد

|  |
| --- |
| let haStr = "Hazzzzah";  let haRegex = /change/; // Change this line  let result = haRegex.test(haStr); |

## تعداد دقیق مطابقت ها را مشخص کنید

با استفاده از {} می توانید تعداد پایین و بالای الگوها را با مشخصه های کمیت مشخص کنید. بعضی اوقات فقط تعداد مشخصی مطابقت را می خواهید. برای مشخص کردن تعداد مشخصی از الگوها ، فقط باید همان یک عدد را بین {} داشته باشید.

به عنوان مثال ، برای مطابقت با کلمه "hah" با حرفa 3 بار ، regex شما برابر با /ha{3}h/. خواهد بود.

|  |
| --- |
| let A4 = "haaaah";  let A3 = "haaah";  let A100 = "h" + "a".repeat(100) + "h";  let multipleHA = /ha{3}h/;  multipleHA.test(A4); // Returns false  multipleHA.test(A3); // Returns true  multipleHA.test(A100); // Returns false |

تمرین3-27: regex ، timRegex را تغییر دهید تا با کلمه "Timber" فقط در صورتی که دارای چهار حرف m باشد مطابقت داشته باشید.

|  |
| --- |
| let timStr = "Timmmmber";  let timRegex = /change/; // Change this line  let result = timRegex.test(timStr); |

## بررسی همه یا هیچ کدام

گاهی اوقات الگوهای مورد نظر برای جستجو ممکن است بخش هایی از آن وجود داشته باشد یا نباشد. با این وجود ، بررسی این موارد مهم است. شما می توانید وجود یک عنصر را با علامت سؤال؟ را مشخص کنید ، این صفر یا یکی از عناصر قبلی را بررسی می کند. می توانید شما می نوانید به این نماد به عنوان گفتن عنصر قبلی اختیاری است فکر کنید.

به عنوان مثال ، در انگلیسی آمریکایی و بریتانیایی اختلافات کمی وجود دارد و شما می توانید از علامت سؤال برای هماهنگ کردن هر دو هجی استفاده کنید.

|  |
| --- |
| let american = "color";  let british = "colour";  let rainbowRegex= /colou?r/;  rainbowRegex.test(american); // Returns true  rainbowRegex.test(british); // Returns true |

تمرین3-28: regex ، favRegex را تغییر دهید تا مطابق با نسخه انگلیسی آمریکایی کلمه ((favorite و انگلیسی بریتانیایی ((favoriteمطابقت داشته باشد.

|  |
| --- |
| let favWord = "favorite";  let favRegex = /change/; // Change this line  let result = favRegex.test(favWord); |

## Lookahead مثبت و منفی

Lookahead ها الگوهایی هستند که به جاوا اسکریپت می گویند که در رشته شما به جلو نگاه کند تا در ادامه الگوهای دیگر را بررسی کند. این می تواند زمانی مفید باشد که می خواهید چندین الگوی مختلف را در همان رشته جستجو کنید. دو نوع lookahead وجود دارد: lookahead مثبت و lookahead منفی.

Lookahead مثبت به این اطمینان خواهد رسید که عنصر الگوی جستجو در آنجا باشد ، اما در واقع با آن مطابقت نخواهد داشت. یک lookahead مثبت به عنوان ?=...) )در جایی که ... قسمت موردنیاز همسان نیست استفاده می شود.

از طرف دیگر ، یک Lookahead منفی به این اطمینان خواهد رسید که عنصر موجود در الگوی جستجو وجود نداشته باشد. از Lookahead منفی به عنوان (?!...) استفاده می شود که در آن ... الگویی است که شما نمی‌خواهید آنجا باشد. در صورت عدم نمایش قسمت Lookahead منفی ، بقیه الگو بازگردانده می شوند.

Lookaheads کمی گیج کننده است ، اما برخی از مثال ها به شما کمک می کنند.

|  |
| --- |
| let quit = "qu";  let noquit = "qt";  let quRegex= /q(?=u)/;  let qRegex = /q(?!u)/;  quit.match(quRegex); // Returns ["q"]  noquit.match(qRegex); // Returns ["q"] |

استفاده عملی تر از lookahead هااین است که دو یا چند الگوی را در یک رشته بررسی کنید. در اینجا یک چک کننده رمزعبور ساده (ساده لوحانه) وجود دارد که به دنبال 3 تا 6 کاراکتر و حداقل یک عدد است:

تمرین3-29: در pwRegex از lookaheadها استفاده کنید تا با گذرواژه هایی که بیش از 5 کاراکتر دارند ، تطبیق دهید ، با اعداد شروع نشود و دو رقم متوالی داشته باشند.

|  |
| --- |
| let sampleWord = "astronaut";  let pwRegex = /change/; // Change this line  let result = pwRegex.test(sampleWord); |

## بررسی ترکیبی گروهی از کاراکتر ها

بعضی اوقات می خواهیم گروه های کاراکتر را با استفاده از یک عبارت منظم بررسی کنیم و باری دستیابی به آن که از پرانتز استفاده می کنیم ().

اگر می خواهید Penguinیا Pumpkin را در رشته پیدا کنید ، می توانید از عبارت منظم زیر استفاده کنید: /P(engu|umpk)in/g.

سپس با استفاده از متد test() تست کنید که آیا گروه‌های رشته مورد نظر در رشته آزمون قرار دارند یا خیر.

|  |
| --- |
| let testStr = "Pumpkin";  let testRegex = /P(engu|umpk)in/;  testRegex.test(testStr);  // Returns true |

تمرین3-30: regex را طوری تنظیم کنید که نام‌های Franklin Roosevelt یا Eleanor Roosevelt را به شکلی حساس به حروف کوچک و بزرگ بررسی کند و برای نام های میانی امتیازاتی ایجاد کند.

سپس کد را طوری تنظیم کنید که regexای که ایجاد کرده اید در مقابل myString بررسی شود و بسته به اینکه آیا regex مطابقت دارد ، trueیا false برگردانده می شود.

|  |
| --- |
| let myString = "Eleanor Roosevelt";  let myRegex = /False/; // Change this line  let result = false; // Change this line  // After passing the challenge experiment with myString and see how the grouping works |

## استفاده مجدد از الگوها با استفاده از capture groups

برخی از الگوهای مورد نظر شما چندین بار در یک رشته اتفاق می افتد. تکرار دستی آن regex بی فایده است. یک روش بهتر برای مشخص کردن وقتی که شماچندین زیررشته تکراری در رشته خود دارید وجود دارد.

می توانید با استفاده از capture groups ،زیر رشته های تکراری را جستجو کنید. پرانتز ، (و)، برای یافتن زیر رشته های تکراری استفاده می شود. شما regex الگویی را که تکرار می شود بین پرانتز ها قرار می دهید.

برای مشخص کردن این که رشته تکراری در کجا ظاهر می شود ، از یک backslash (\) و سپس یک عدد استفاده می کنید. این عدد از 1 شروع می شود و با هر capture groups که استفاده می کنید افزایش می‌یابد. نمونه ای برای مطابقت با گروه اول \1 خواهد بود.

مثال زیر با هر کلمه ای که دو بار با یک فضای خالی از هم جدا شود مطابقت دارد:

|  |
| --- |
| let repeatStr = "regex regex";  let repeatRegex = /(\w+)\s\1/;  repeatRegex.test(repeatStr); // Returns true  repeatStr.match(repeatRegex); // Returns ["regex regex", "regex"] |

با استفاده از متد .match() روی یک رشته ، آرایه ای با رشته ای که مطابقت دارد ، به همراه capture groups آن برمی گرداند.

تمرین3-31: از capture groups در reRegex برای مطابقت با اعدادی استفاده کنید که فقط سه بار در یک رشته تکرار می شوند ، هر کدام با یک فاصله جدا می شوند.

|  |
| --- |
| let repeatNum = "42 42 42";  let reRegex = /change/; // Change this line  let result = reRegex.test(repeatNum); |

## استفاده از گروه های capture برای جستجو و جایگزینی

جستجو مفید است با این حال ، وقتی متنی را که مطابقت دارد تغییر می دهد (یا جایگزین می کند) می توانید جستجو را حتی قدرتمندتر کنید. می توانید متن را با استفاده از .replace() روی یک رشته جستجو کرده و جایگزین کنید. ورودی برای .replace()ابتدا الگوی regex است که می خواهید جستجو کنید. پارامتر دوم رشته ای است که جایگزین مطابقت یا یک تابع برای انجام کاری می شود.

|  |
| --- |
| let wrongText = "The sky is silver.";  let silverRegex = /silver/;  wrongText.replace(silverRegex, "blue");  // Returns "The sky is blue." |

همچنین می توانید به گروه های capture در رشته جایگزینی با علائم دلار ($) دسترسی پیدا کنید.

|  |
| --- |
| "Code Camp".replace(/(\w+)\s(\w+)/, '$2 $1');  // Returns "Camp Code" |

تمرین3-32: با استفاده از سه گروه capture ، regex ، fixRegex را بنویسید که هر کلمه را در رشته "one two three" جستجو می کند. سپس متغیر replaceText را به روز کنید تا "one two three" را با رشته "three two one" جایگزین کنید و نتیجه را به متغیر result اختصاص دهید. اطمینان حاصل کنید که از گروه های capture در رشته جایگزینی با استفاده از نحو علامت دلار ($) استفاده می کنید.

|  |
| --- |
| let str = "one two three";  let fixRegex = /change/; // Change this line  let replaceText = ""; // Change this line  let result = str.replace(fixRegex, replaceText); |

## Whitespace را از ابتدا و انتها حذف کنید

گاهی اوقات کاراکترهای فضای سفید در اطراف رشته ها مورد نظر نیستند اما وجود دارند. پردازش معمولی رشته ها فضای سفید در شروع و انتهای آن را حذف می کند.

تمرین3-33: یک regex بنویسید و از متدهای رشته مناسب برای پاک کردن فضای سفید در آغاز و انتهای رشته ها استفاده کنید.

توجه: متد String.prototype.trim() در اینجا کار خواهد کرد ، اما شما باید این چالش را با استفاده از عبارات منظم تکمیل کنید.

|  |
| --- |
| let hello = "   Hello, World!  ";  let wsRegex = /change/; // Change this line  let result = hello; // Change this line |

# 4. اشکال‌زدایی

## آشنایی با چالش های اشکال زدایی

اشکال زدایی ابزاری ارزشمند و (متاسفانه) ضروری برای برنامه نویسان است. این فاز آزمایش بررسی اینکه کد شما طبق برنامه کار می کند ، و کشف اینکه آن کار نمی کند دنبال می شود. اشکال زدایی فرایند یافتن دقیقاً همان چیزی است که کار نمی کند اصلاح آن است. پس از گذراندن وقت برای ایجاد یک بلوک درخشان از کد ، درک این موضوع دشوار است که ممکن است خطایی داشته باشد. این مشکلات به طور کلی به سه فرم ارائه می‌شود:

1. خطاهای نحوی که از اجرای برنامه جلوگیری می کنند
2. خطاهای زمان اجرا هنگامی که اجرا کد با شکست مواجه می‌شود یا عملکرد غیر منتظره ای دارد.
3. خطاهای معنایی (یا منطقی) وقتی کد آنچه را که منظورش است انجام نمی دهد.

ویرایشگرهای مدرن کد (و تجربه) می توانند به شناسایی خطاهای نحوی کمک کنند. یافتن خطاهای معنایی و زمان اجرا سخت تر است. آنها ممکن است باعث خراب شدن برنامه شما شوند ، آن را برای همیشه اجرا کنند یا به نتیجه نادرست منجر شوند. به اشکال زدایی به عنوان سعی در درک دلیل کد شما به روشی که رفتار می‌کند فکر کنید. نمونه ای از خطای نحو - که اغلب توسط ویرایشگر کد شناسایی می شود:

|  |
| --- |
| funtcion willNotWork(    console.log("Yuck");  }  // "function" keyword is misspelled and there's a missing parenthesis |

اینجا مثالی از خطای زمان اجرا وجود دارد - اغلب در حین اجرای برنامه تشخیص داده می شود:

|  |
| --- |
| function loopy() {    while(true) {      console.log("Hello, world!");    }  }  // Calling loopy starts an infinite loop, which may crash your browser |

نمونه ای از خطای معنایی - که اغلب پس از آزمایش کد خروجی تشخیص داده می شود:

|  |
| --- |
| function calcAreaOfRect(w, h) {    return w + h; // This should be w \* h  }  let myRectArea = calcAreaOfRect(2, 3);  // Correct syntax and the program executes, but this gives the wrong answer |

اشکال زدایی ناامید کننده است ، اما به شما در توسعه (و دنبال کردن) یک روش گام به گام برای مرور کد شما کمک می کند. به این معنی که مقادیر متوسط ​​و انواع متغیرها را بررسی کنید تا ببینید آیا آنها همان چیزی هستند که باید باشند. می توانید با یک فرآیند ساده حذف شروع کنید.

به عنوان مثال ، اگر تابع A کار کند و آنچه را که قرار است برگرداند ، سپس ممکن است تابع B با مشکل مواجه شود. یا از وسط بررسی مقادیر را در یک بلوک کد شروع کنید تا سعی کنید فضای جستجو را به نصف کاهش دهید. مشکلی که در یک نقطه وجود دارد ،نشان می‌دهد اشکال در نیمه اول کد است. اگر اینطور نیست ، احتمالاً در نیمه دوم است.

در این بخش چند ابزار مفید برای پیدا کردن اشکالات و برخی از اشکال متداول آنها وجود دارد. خوشبختانه ، اشکال زدایی یک مهارت قابل یادگیری است که فقط نیاز به کمی صبر و تمرین برای تسلط دارد.

## برای بررسی مقدار متغیر از کنسول JavaScript استفاده کنید

Chrome و Firefox هر دو دارای کنسول‌های JavaScript عالی هستند ، همچنین برای اشکال زدایی در JavaScript ، به عنوان DevTools شناخته می‌شوند.

می توانید ابزارهای توسع دهنده را در منوی Chrome یا کنسول وب در منوی Firefox پیدا کنید. اگر از مرورگر دیگری یا تلفن همراه استفاده می کنید ، توصیه می کنیم به نسخه دسکتاپ Firefox یا Chrome بروید.

متد console.log() ، که آنچه را در داخل پرانتزهای آن است را در کنسول چاپ می‌کند، به احتمال زیاد مفیدترین ابزار اشکال زدایی خواهد بود. قرار دادن آن در نقاط استراتژیک در کد شما می تواند مقادیر متوسط ​​متغیرها را به شما نشان دهد. این تمرین خوبی است که قبل از بررسی آنچه که در آن است ، تصوری از خروجی داشته باشید. داشتن نقاط چک برای مشاهده وضعیت محاسبات در کل کد شما به کوچکتر شدن محل مشکل کمک می کند.

در اینجا مثالی برای چاپ 'Hello world!' به کنسول هست:

console.log('Hello world!');

تمرین4-1: برای چاپ مقدار متغیر a که در hdk کد ذکر شده است ، از متد console.log() استفاده کنید.

|  |
| --- |
| let a = 5;  let b = 1;  a++;  let sumAB = a + b;  console.log(sumAB); |

## درک تفاوت بین کنسول freeCodeCamp و مرورگر

شاید متوجه شده باشید که برخی از چالش های JavaScript ، freeCodeCamp شامل کنسول مخصوص به خود هستند. این کنسول کمی متفاوت از کنسول مرورگر شما در آخرین چالش رفتار می کند.

چالش زیر به منظور برجسته کردن تفاوت اصلی بین کنسول freeCodeCamp و کنسول مرورگر شما است. هنگامی که جاوا اسکریپت معمولی را اجرا می کنید ، کنسول مرورگر دستور console.log() شما را با تعداد دقیق دفعاتی که فراخوانی شده نمایش می دهد.

کنسول freeCodeCamp پس از اینکه ویرایشگر متوجه تغییر در اسکریپت و همچنین در حین تست می شود ، دستورات console.log() شما را مدت کوتاهی چاپ می کند.

کنسول freeCodeCamp قبل از اجرای تست پاک می شود و برای جلوگیری از هرزنامه ، گزارش ها فقط را در طول تست اول چاپ می کنید (برای استثنائات به یادداشت زیر مراجعه کنید).

اگر مایل هستید برای هر تست ، هر گزارش را مشاهده کنید ، تست ها را اجرا کنید و کنسول مرورگر را باز کنید. اگر ترجیح می دهید از کنسول مرورگر استفاده کنید و می خواهید از کنسول freeCodeCamp تقلید کنید ، قبل از هرگونه فراخوانی کنسول دیگر ، console.clear() را قرار دهید تا کنسول مرورگر را پاک کنید.

توجه: console.log ها را در داخل توابع در کنسول freeCodeCamp چاپ کنید هر زمان که این توابع فراخوانی شود ، این می تواند به توابع اشکال زدایی که هنگام تست فراخوانی می شوند ، کمک کند.

تمرین4-2: ابتدا برای وارد کردن متغیر output از console.log استفاده کنید. سپس ، از console.clear برای پاک کردن کنسول مرورگر استفاده کنید.

|  |
| --- |
| // Open your browser console.  let output = "Get this to log once in the freeCodeCamp console and twice in the browser console";  // Use console.log() to print the output variable.  // Run the tests to see the difference between the two consoles.  // Now, add console.clear() before your console.log() to clear the browser console, and pass the tests. |

## برای بررسی نوع متغیر از typeof استفاده کنید

برای بررسی ساختار داده یا نوع متغیر می توانید از typeof استفاده کنید. این در اشکال زدایی هنگام کار با انواع مختلف داده ها مفید است. اگر فکر می کنید دو عدد اضافه می کنید ، اما یکی در واقع یک رشته است ، نتایج می تواند غیر منتظره باشد. خطاهای نوع می توانند در محاسبات یا فراخوانی های تابع کمین کنند. مراقب باشید به خصوص هنگام دسترسی و کار با داده های خارجی در قالب یک هدف JavaScript Object Notation شی (JSON).

در اینجا چند مثال با استفاده از typof آورده شده است:

|  |
| --- |
| console.log(typeof ""); // outputs "string"  console.log(typeof 0); // outputs "number"  console.log(typeof []); // outputs "object"  console.log(typeof {}); // outputs "object" |

جاوا اسکریپت شش نوع داده اولیه (تغییر ناپذیر) را تشخیص می دهد: بولی ،Boolean, Null, Undefined, Number, String و Symbol (جدید با ES6) و یک نوع برای موارد قابل تغییر: Object. توجه داشته باشید که در JavaScript ، آرایه ها از نظر فنی نوعی شیء هستند.

تمرین4-3: برای بررسی typeof هر دو متغیر seven وthree در کد ، دو دستور console.log() اضافه کنید.

|  |
| --- |
| let seven = 7;  let three = "3";  console.log(seven + three); |

## نام متغیرها و توابع را غلط املایی بگیرید

متدهای console.log() و typeof دو روش اصلی برای بررسی مقادیر متوسط ​​و انواع خروجی برنامه هستند. اکنون زمان آن رسیده است که به اشکال متداولی که خطاها می‌گیرند وارد شوید. یک مسئله در سطح نحوی که تایپ کننده های سریع می توانند با آن مرتکب شوند خطای غلط املایی است.

کاراکترهای جا به جا ، گمشده یا حروف بزرگ و کوچک نادرست در یک متغیر یا نام تابع باعث می شوند مرورگر به دنبال شیء ای باشد که وجود ندارد - و به شکلی از خطای مرجع شکایت کند. متغیرها و نام‌های تابع JavaScript حساس به حروف کوچک و بزرگ هستند.

تمرین4-4: دو خطای غلط املایی را در کد برطرف کنید تا محاسبه netWorkingCapital کار کند.

|  |
| --- |
| let receivables = 10;  let payables = 8;  let netWorkingCapital = recievables - payable;  console.log(`Net working capital is: ${netWorkingCapital}`); |

## گرفتن پرانتز ، براکت ، کروشه و نقل قول بسته نشده

یکی دیگر از خطاهای نحوی که باید از آن آگاهی داشته باشید این است که همه پرانتزهای باز ، براکت ها ، کروشه ها و نقل قول ها دارای یک جفت بسته هستند. فراموش کردن یک قطعه معمولاً هنگام ویرایش کد موجود و درج آیتم ها در یکی از انواع جفت ها اتفاق می افتد. همچنین ، هنگام قراردادن بلوک کد درون بلوک دیگری ، مانند اضافه کردن تابع پاسخ به عنوان یک آرگومان برای یک متد ، مراقب باشید.

یکی از راه های جلوگیری از این اشتباه این است که به محض تایپ کردن کاراکتر بازشده ، آن را ببندید ، سپس مکان نما را بین آنها بکشید و کدنویسی را ادامه دهید. خوشبختانه ، بیشتر ویرایشگرهای مدرن کد ، نیمه دوم این جفت ها را بطور خودکار تولید می کنند.

تمرین4-5: خطاهای دو جفت را در کد رفع کنید.

|  |
| --- |
| l let myArray = [1, 2, 3;  let arraySum = myArray.reduce((previous, current =>  previous + current);  console.log(`Sum of array values is: ${arraySum}`); |

## گرفتن خطای ترکیب نقل فول های تکی و دوتایی

جاوا اسکریپت اجازه می دهد تا از نقل قول های تکی (') و مضاعف (") برای اعلان یک رشته استفاده کنید. تصمیم گیری در مورد اینکه کدام یک از آنها به طور کلی استفاده شود ، با برخی موارد استثناء به به ترجیح شخصی بستگی دارد.

داشتن دو گزینه بسیار مناسب است وقتی یک رشته دارای اختصارات یا متن دیگری است که به نقل از دیگران است. فقط مراقب باشید رشته را خیلی زود ببندید ، که باعث خطای نحو می شود.

در اینجا چند نمونه از مخلوط کردن نقل قول ها آورده شده است:

|  |
| --- |
| // These are correct:  const grouchoContraction = "I've had a perfectly wonderful evening, but this wasn't it.";  const quoteInString = "Groucho Marx once said 'Quote me as saying I was mis-quoted.'";  // This is incorrect:  const uhOhGroucho = 'I've had a perfectly wonderful evening, but this wasn't it.'; |

البته ، استفاده از تنها یک سبک نقل قول ، اشکالی ندارد. می توانید با استفاده از کاراکتر فرار (\)از نقل قول های داخل رشته فرار کنید:

|  |
| --- |
| // Correct use of same quotes:  const allSameQuotes = 'I\'ve had a perfectly wonderful evening, but this wasn\'t it.'; |

تمرین4-6: رشته را اصلاح کنید تا از علامت های نقل قول مختلف برای مقدار href استفاده کند یا از موارد موجود فرار کند. علائم نقل قول دوتایی را در کل رشته نگه دارید.

|  |
| --- |
| let innerHtml = "<p>Click here to <a href="#Home">return home</a></p>";  console.log(innerHtml); |

## گرفتن خطای استفاده از عملگر انتساب به جای عملگر برابری

برنامه های شاخه ، یعنی برنامه هایی که در صورت برقرار شدن شرایط خاص کارهای مختلفی را انجام می‌دهند ، به دستورات if, else if و else جاوا اسکریپت متکی هستند. این شرط گاهی به شکل تست صورت می گیرد که آیا نتیجه برابر با یک مقدار است.

به این منطق (حداقل به انگلیسی) به عنوان "اگر x برابر با y باشد ، پس از آن ..." گفته می شود که به معنای واقعی کلمه می تواند با استفاده از = ، یا عملگر مساوی ، به کد تبدیل شود. این منجر به جریان کنترل غیر منتظره در برنامه شما می شود.

مطابق با چالش های قبلی ، عملگر انتساب (=) در جاوا اسکریپت مقداری را به یک نام متغیر اختصاص می دهد. و عملگرهای == و === برابری را بررسی می کنند (آزمون‌های سه گانه === برای برابری دقیق ، یعنی هم ارزش و نوع یکسان هستند).

کد زیر x را با عدد 2 اختصاص می دهد ، که true ارزیابی می‌شود. تقریباً هر مقداری در جاوا اسکریپت true ارزیابی می‌شود ، به جز آنچه که به مقادیر "جعلی" معروف است: false ، 0 ، "" (یک رشته خالی) ، NaN ، undefinedو null.

|  |
| --- |
| let x = 1;  let y = 2;  if (x = y) {    // this code block will run for any value of y (unless y were originally set as a falsy)  } else {    // this code block is what should run (but won't) in this example |

تمرین4-7: شرط را اصلاح کنید تا برنامه شاخه مناسب را اجرا کند و مقدار مناسب به result اختصاص ‌یابد.

|  |
| --- |
| let x = 7;  let y = 9;  let result = "to come";  if(x = y) {    result = "Equal!";  } else {    result = "Not equal!";  }  console.log(result); |

## گرفتن خطای وجود نداشتن پرانتز باز و بسته پس از فراخوانی یک تابع

هنگامی که یک تابع یا متد هیچ آرگومانی ندارد ، ممکن است هنگام فراخوانی ، پرانتزهای (خالی) باز و بسته را فراموش کنید. اغلب اوقات نتیجه یک فراخوانی تابع در یک متغیر برای استفاده دیگر در کد شما ذخیره می‌شود. این خطا را می توان با ورود مقادیر متغیر (یا انواع آنها) به کنسول و دیدن اینکه به جای برگرداند مقدار مورد انتظار تابع، به یک مرجع تابع تنظیم کرده است ، پیدا کرد.

متغیرهای موجود در مثال زیر متفاوت هستند:

|  |
| --- |
| function myFunction() {    return "You rock!";  }  let varOne = myFunction; // set to equal a function  let varTwo = myFunction(); // set to equal the string "You rock!" |

تمرین4-8: کد را اصلاح کنید تا متغیر result با مقدار برگشتی از فراخوانی تابع getNine تنظیم شود.

|  |
| --- |
| function getNine() {    let x = 6;    let y = 3;    return x + y;  }  let result = getNine;  console.log(result); |

## گرفتن خطای ارسال آرگومان ها به ترتیب اشتباه به هنگام فراخوانی تابع

ادامه بحث در مورد فراخوانی توابع ، اشکال بعدی که باید مراقب باشید وقتی آرگومان های یک تابع به ترتیب نادرست ارائه می شود. اگر آرگومان ها انواع مختلفی دارند ، مانند تابعی که انتظار آرایه و یک عدد صحیح را دارد ، احتمالاً این باعث خطای زمان اجرا را می شود. اگر آرگومان ها یک نوع هستند (به عنوان مثال تمام عدد صحیح) ، منطق کد معنی ندارد. برای جلوگیری از این مسائل ، حتماً تمام آرگومان های لازم را تهیه کنید.

تمرین4-9: تابع ngritToPower پایه ای را به توان می رساند. متأسفانه ، آن به درستی فراخوانی نشده است - کد را تصحیح کنید تا مقدارpower انتظار می رود8 باشد.

|  |
| --- |
| function raiseToPower(b, e) {    return Math.pow(b, e);  }  let base = 2;  let exp = 3;  let power = raiseToPower(exp, base);  console.log(power); |

## گرفتن توقف با یگ خطا در هنگام استفاده ازایندکس اتفاق می افتد

خاموش شدن با یک خطا (که گاهی اوقات OBOE نیز گفته می شود) زمانی که شما می خواهید یک شاخص خاص از یک رشته یا آرایه را هدف قرار دهید (برای برش یا دسترسی به یک بخش) ، یا هنگام پیمایش بین ایندکس های آنها. ایندکس گذاری جاوا اسکریپت از صفر شروع می شود ، و نه یک ، به این معنی که آخرین ایندکس همیشه یکی از طول آرایه کمتر است. اگر سعی کنید به یک ایندکس برابر با طول دسترسی پیدا کنید ، این برنامه ممکن است خطای مرجع " خارج از محدوده" را نشان دهد یا undefined را چاپ کند.

وقتی از متد های رشته ای یا آرایه ای استفاده می کنید که دامنه های ایندکس را به عنوان آرگومان در نظر می‌گیرند ، به خواندن اسناد کمک می کند و می فهمید که آیا آن ها شامل هستند (آیتم موجود در ایندکس مشخص شده بخشی از چیزی است که بازگردانده می شود) یا نه. در اینجا چند مثال از خاموش کردن با یک خطا آورده شده است:

|  |
| --- |
| let alphabet = "abcdefghijklmnopqrstuvwxyz";  let len = alphabet.length;  for (let i = 0; i <= len; i++) {    // loops one too many times at the end    console.log(alphabet[i]);  }  for (let j = 1; j < len; j++) {    // loops one too few times and misses the first character at index 0    console.log(alphabet[j]);  }  for (let k = 0; k < len; k++) {    // Goldilocks approves - this is just right    console.log(alphabet[k]);  } |

تمرین4-10: دو خطای ایندکس گذاری را در تابع زیر برطرف کنید تا تمام اعداد 1 تا 5 در کنسول چاپ شوند.

|  |
| --- |
| function countToFive() {    let firstFive = "12345";    let len = firstFive.length;    for (let i = 1; i <= len; i++) {      console.log(firstFive[i]);    }  }  countToFive(); |

## هگام مقداردهی مجدد متقیر ها داخل حلقه احتیاط کنید

گاهی اوقات ذخیره اطلاعات افزایش شمارنده ها، یا تنظیم مجدد متغیرها در یک حلقه لازم است. مسئله بالقوه زمانی است که متغیرها یا باید مجدداً مورد استفاده قرار گیرند ، و اینگونه نیستند ، یا برعکس. این خصوصاً خطرناک است اگر به طور اتفاقی متغیر مورد استفاده را برای شرط ترمینال تنظیم مجدد کنید ، باعث یک حلقه بی نهایت می شود.

چاپ مقادیر متغیر با هر چرخه حلقه خود با استفاده از console.log() می تواند رفتار اشکال مربوط به تنظیم مجدد یا عدم تنظیم مجدد متغیر را کشف کند.

تمرین4-11: تابع زیر قرار است یک آرایه دو بعدی را با m ردیف وn ستون هایی از صفرها ایجاد کند. متأسفانه ، خروجی مورد انتظار را تولید نمی کند زیرا متغیر row در حلقه بیرونی مجدداً تنظیم نمی شود (به یک آرایه خالی باز می گردد). کد را اصلاح تا یک آرایه صحیح 3x2 از صفرها، که اینگونه نظر می رسد [[0, 0], [0, 0], [0, 0]] باشد.

|  |
| --- |
| function zeroArray(m, n) {    // Creates a 2-D array with m rows and n columns of zeroes    let newArray = [];    let row = [];    for (let i = 0; i < m; i++) {      // Adds the m-th row into newArray      for (let j = 0; j < n; j++) {        // Pushes n zeroes into the current row to create the columns        row.push(0);      }      // Pushes the current row, which now has n zeroes in it, to the array      newArray.push(row);    }    return newArray;  }  let matrix = zeroArray(3, 2);  console.log(matrix); |

## جلوگیری از حلقه های نامتناهی با یک شرط پایانی معتبر

موضوع آخر حلقه نامحدود مخوف است. حلقه ها ابزارهای خوبی برای زمانی که شما نیاز دارید بلوک کد برای تعداد محدودی معینی یا تا زمان برقراری یک شرط اجرا شود هستند ، به برنامه خود کمک می کنید ، اما آنها به یک شرط پایانی معتبر ل نیاز دارند که حلقه را خاتمه دهد. حلقه های بی نهایت احتمالاً مرورگر را مسدود یا خراب می کنند و باعث خرابی اجرای عمومی برنامه می شوند ، که هیچ کس نمی خواهد.

مثالی از حلقه نامتناهی در معرفی این بخش وجود دارد - هیچ شرط معتبری برای خارج شدن از حلقه while در داخل حلقه loopy() وجود ندارد. این تابع را فراخوانی نکنید!

|  |
| --- |
| function loopy() {    while(true) {      console.log("Hello, world!");    }  } |

این وظیفه برنامه نویس است که اطمینان حاصل کند که شرط معتبر ، که به برنامه می گوید چه موقع باید از کد حلقه خارج شود ، سرانجام رسیده است. یک خطا افزایش یا کاهش یک متغیر شمارنده در جهت اشتباه از شرط معتبر است. خطای دیگر این است که بجای افزایش یا کاهش آن ، متغیر شمارنده یا ایدکس را درون کد حلقه تنظیم مجدد کنید.

تمرین4-12: تابع myFunc() حاوی یک حلقه نامحدود است زیرا شرط نهایی i != 4 هرگز false ارزیابی نمی‌شود(و حلقه را می شکند) - i هر ارسال را با 2 افزایش می دهد و از بالای 4 پرش می کند زیرا i در شروع فرد است. عملگر مقایسه را در شرط معتبر اصلاح کنید تا حلقه فقط برای i کمتر یا مساوی 4 باشد.

|  |
| --- |
| function myFunc() {    for (let i = 1; i != 4; i += 2) {      console.log("Still going!");    }  } |

# 5. ساختان داده‌های اصلی

## آشنایی با چالش های ساختان داده اصلی

داده ها را می توان به روش های مختلف ، چه در JavaScript و چه به زبان های دیگر ، ذخیره و به آنها دسترسی پیدا کرد. در این بخش نحوه دستکاری آرایه ها و همچنین دسترسی و کپی کردن اطلاعات موجود در آنها به شما یاد داده می‌دهد. همچنین نحوه دستکاری و دسترسی به داده ها را در اشیاء جاوا اسکریپت ، با استفاده از نشانه های نقطه و براکت ، به شما یاد می دهد. هنگامی که با این بخش کار خود را تمام کردید ، باید خصوصیات و تفاوت های اساسی بین آرایه ها و اشیاء و همچنین نحوه انتخاب هرکدام را برای یک هدف معین درک کنید.

## برای ذخیره مجموعه ای از داده ها از یک آرایه استفاده کنید

در زیر نمونه ای از ساده ترین اجزای ساختار داده آرایه است. این به عنوان یک آرایه یک بعدی شناخته شده است ، به این معنی که فقط یک سطح دارد ، یا این که هیچ آرایه دیگری در داخل آن جای ندارد. توجه کنید که این برنامه شامل بولین ها، رشته ها و اعداد در میان سایر داده های معتبر JavaScript است:

|  |
| --- |
| let simpleArray = ['one', 2, 'three', true, false, undefined, null];  console.log(simpleArray.length);  // logs 7 |

همه آرایه ها دارای یک خاصیت طول هستند که مطابق شکل فوق با نحو Array.length به راحتی قابل دسترسی است. اجزای پیچیده تر یک آرایه را می توان در زیر مشاهده کرد. این به عنوان یک آرایه چند بعدی یا آرایه ای معروف است که شامل آرایه های دیگری است. توجه کنید که این آرایه شامل اشیاء جاوا اسکریپت نیز می باشد ، که ما در بخش بعدی آنها را با دقت مورد بررسی قرار خواهیم داد ، اما در حال حاضر ، تمام آنچه شما باید بدانید این است که آرایه ها همچنین قادر به ذخیره اشیاء پیچیده هستند.

|  |
| --- |
| let complexArray = [    [      {        one: 1,        two: 2      },      {        three: 3,        four: 4      }    ],    [      {        a: "a",        b: "b"      },      {        c: "c",        d: "d"      }    ]  ]; |

تمرین 5-1: ما متغیری به نام yourArray تعریف کرده ایم. دستور را با اختصاص آرایه ای به طول حداقل 5 عنصر به متغیر yourArray تکمیل کنید. آرایه شما باید حاوی حداقل یک رشته ، یک شماره و یک بولین باشد.

|  |
| --- |
| let yourArray; |

## با استفاده از نماد براکت به محتوای آرایه دسترسی پیدا کنید

ویژگی اصلی هر ساختار داده ، البته ، امکان ذخیره نه تنها داده ها ، بلکه امکان بازیابی آن داده ها با دستور است. بنابراین ، اکنون که یاد گرفتیم که چگونه یک آرایه ایجاد کنیم ، بیایید به این فکر کنیم که چگونه می‌توانیم به اطلاعات آن آرایه دسترسی داشته باشیم.

وقتی یک آرایه ساده را مطابق شکل زیر تعریف می کنیم ، 3 مورد در آن وجود دارد:

|  |
| --- |
| let ourArray = ["a", "b", "c"]; |

در یک آرایه ، هر عنصر آرایه ای دارای یک ایندکس است. این ایندکس به عنوان موقعیت آن عنصر در آرایه ، و نحوه ارجاع به آن می شود. با این حال ، توجه به این نکته ضروری است که آرایه های جاوا اسکریپت دارای ایندکس از صفر هستند ، بدین معنی که اولین عنصر یک آرایه در واقع در موقعیت صفر است نه یک.

برای بازیابی یک عنصر از یک آرایه می توانیم یک ایندکس را در براکت ها محصور کنیم و آن را در انتهای یک آرایه یا معمولاً به یک متغیر اضافه کنیم که به یک شی آرایه ارجاع می دهد. این به عنوان نماد براکت معروف است. به عنوان مثال ، اگر می خواهیم "a" را از ourArray بازیابی کنیم و آن را به یک متغیر اختصاص دهیم ، می توانیم با کد زیر این کار را انجام دهیم:

|  |
| --- |
| let ourVariable = ourArray[0];  // ourVariable equals "a" |

با استفاده از نماد براکت ، اکنون عنصررا درایندکس 1از "b" در"not b anymore" تنظیم کرده ایم.

تمرین 5-2: برای تکمیل این چالش ، علاوه بر "b" ، موقعیت 2 (ایندکس 1) myArray را بر روی هر چیزی که می خواهید قرار دهید.

|  |
| --- |
| let myArray = ["a", "b", "c", "d"];  console.log(myArray); |

## اضافه کردن عناصر به یک آرایه با push() وunshift()

طول آرایه مانند انواع داده های موجود در آن ثابت نیست. آرایه ها را می توان با طول هر تعداد عنصر تعریف کرد و عناصر با گذشت زمان می توانند اضافه یا حذف شوند. به عبارت دیگر ، آرایه ها قابل تغییر هستند. در این چالش ، ما به دو متد خواهیم پرداخت که با استفاده از آن ها می توانیم یک آرایه را اصلاح کنیم: Array.push() و Array.unshift().

هر دو متد یک یا چند عنصر را به عنوان پارامتر در نظر می گیرند و آن عناصر را به آرایه ای که با متد فراخوانی می شود اضافه می کنند; متد push () عناصر را به انتهای یک آرایه اضافه می کند ، و unshift() عناصر را به ابتدا اضافه می کند. موارد زیر را در نظر بگیرید:

|  |
| --- |
| let twentyThree = 'XXIII';  let romanNumerals = ['XXI', 'XXII'];  romanNumerals.unshift('XIX', 'XX');  // now equals ['XIX', 'XX', 'XXI', 'XXII']  romanNumerals.push(twentyThree);  // now equals ['XIX', 'XX', 'XXI', 'XXII', 'XXIII']Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data. |

تمرین 5-3: ما یک تابع ، mixedNumbers را تعریف کرده ایم که یک آرایه را به عنوان آرگومان به آن می‌فرستیم . تابع را با استفاده از push() و unshift() برای اضافه کردن 'I', 2, 'three' به ابتدای آرایه و 7, 'VIII', 9 به انتهای آن اضافه کنید تا آرایه برگشتی به ترتیب حاوی بازنمودهای اعداد 1-9 باشد.

|  |
| --- |
| function mixedNumbers(arr) {      return arr;  }  console.log(mixedNumbers(['IV', 5, 'six'])); |

## حذف کردن عناصراز یک آرایه با pop() و shift ()

هر دو push() و unshift() متدهای متناظر دارند که تقریباً کاربرد برعکس دارند: pop() و shift(). همانطور که احتمالاً تاکنون حدس زده اید ، به جای اضافه کردن ، pop() یک عنصر را از انتهای یک آرایه حذف می کند ، در حالی که shift() یک عنصر را از ابتدا حذف می کند. تفاوت اصلی بین pop() و shift() و پسر عموی آنها push() و unshift() در این است که هیچ پارامتری را نمی گیرد و هرکدام فقط اجازه می دهند که یک آرایه توسط یک عنصر واحد در یک زمان اصلاح شود. بیا یک نگاهی بیندازیم:

|  |
| --- |
| let greetings = ['whats up?', 'hello', 'see ya!'];  greetings.pop();  // now equals ['whats up?', 'hello']  greetings.shift();  // now equals ['hello'] |

ما همچنین می توانیم مقدار عنصر حذف شده را با روش دیگری مانند این برگردانیم:

|  |
| --- |
| let popped = greetings.pop();  // returns 'hello'  // greetings now equals [] |

تمرین 5-4: ما یک تابع ، popShift را تعریف کرده ایم که یک آرایه را به عنوان یک آرگومان می گیرد و یک آرایه جدید را برمی گرداند. تابع را تغییر دهید ، با استفاده از pop () و shift () ، عناصر اول و آخر آرایه را حذف کنید و عناصر حذف شده را به متغیرهای مربوطه اختصاص دهید ، به طوری که آرایه برگشتی حاوی مقادیر آنها باشد.

|  |
| --- |
| function popShift(arr) {    let popped; // Change this line    let shifted; // Change this line    return [shifted, popped];  }  console.log(popShift(['challenge', 'is', 'not', 'complete'])); |

## حذف عناصر با استفاده از () splice

خوب ، بنابراین ما آموخته ایم که چگونه عناصر را از ابتدا و انتهای آرایه ها با استفاده از shift() و pop() حذف کنیم ، اما اگر بخواهیم یک عنصر را از جایی در وسط حذف کنیم ، چه می شود؟ یا بیش از یک عنصر به طور همزمان حذف شود؟ خوب ، اینجاست که splice() وارد می شود. splice() به ما امکان می دهد همین کار را انجام دهیم: تعداد هر عنصر پی در پی را از هر کجای یک آرایه حذف کنیم.

splice () می تواند تا 3 پارامتر بگیرد ، اما در حال حاضر ، ما فقط بر روی 2 مورد اول متمرکز خواهیم شد. دو پارامتر اول splice()عدد صحیحی هستند که ایندکس ها یا موقعیت ها را از آرایه ای که با splice() خراخوانی می شود ، نشان می دهند. و به یاد داشته باشید ، آرایه ها از ایندکس صفر هستند ، بنابراین برای نشان دادن اولین عنصر یک آرایه ، ما از 0 استفاده می کنیم.

اولین پارامتر splice() ایندکسی از آرایه است که از آن شروع به از بین بردن عناصر می کند ، و پارامتر دوم تعداد عناصر حذف شده را نشان می دهد. مثلا:

|  |
| --- |
| let array = ['today', 'was', 'not', 'so', 'great'];  array.splice(2, 2);  // remove 2 elements beginning with the 3rd element  // array now equals ['today', 'was', 'great'] |

splice() نه تنها آرایه ای را که فراخوانی می شود اصلاح می کند ، بلکه یک آرایه جدید را نیز حاوی مقدار عناصر حذف شده است را برمی گرداند:

|  |
| --- |
| let array = ['I', 'am', 'feeling', 'really', 'happy'];  let newArray = array.splice(3, 2);  // newArray equals ['really', 'happy'] |

تمرین 5-5: ما یک آرایه arrرا مقداردهی کرده ایم. برای حذف کردن عناصر از arr ، از splice()استفاده کنید ، به این ترتیب که فقط عناصری دارد که مجموع آن ها 10 می شود.

|  |
| --- |
| const arr = [2, 4, 5, 1, 7, 5, 2, 1];  console.log(arr); |

## اضافه کردن عناصرآرایه با استفاده از () splice

به یاد داشته باشید در آخرین چالشی که ذکر کردیم splice()می تواند تا سه پارامتر داشته باشد؟ خوب ، می‌توانید از پارامتر سوم ، متشکل از یک یا چند عنصر برای اضافه کردن به آرایه استفاده کنید.

این می تواند برای جابه جایی سریع یک عنصر یا مجموعه ای از عناصر با عنصر دیگر فوق العاده مفید باشد.

|  |
| --- |
| const numbers = [10, 11, 12, 12, 15];  const startIndex = 3;  const amountToDelete = 1;  numbers.splice(startIndex, amountToDelete, 13, 14);  // the second entry of 12 is removed, and we add 13 and 14 at the same index  console.log(numbers);  // returns [ 10, 11, 12, 13, 14, 15 ] |

در اینجا ما با آرایه ای از اعداد شروع می کنیم. سپس موارد زیر را به splice() ارسال می کنیم. ایندکس شروع به حذف عناصر (3) ، تعداد عناصر حذف شده (1) و عناصر (13 ، 14) می شود که در همان ایندکس درج می شوند. توجه داشته باشید که در زیر amountToDelete هر تعداد عنصری می تواند باشد(که با کاما جدا شده اند) ، که هر یک از آنها درج شده اند.

تمرین 5-6: ما یک تابع ، htmlColorNames را تعریف کرده ایم ، که آرایه ای از رنگ های HTML را به عنوان آرگومان می‌گیرد. تابع را با استفاده از splice() اصلاح كنید تا دو عنصر اول آرایه را از حذف کنید و "DarkSalmon" و "BlanchedAlmond" را در مکان های مربوطه آن ها اضافه كنید.

|  |
| --- |
| function htmlColorNames(arr) {    return arr;  }  console.log(htmlColorNames(['DarkGoldenRod', 'WhiteSmoke', 'LavenderBlush', 'PaleTurquoise', 'FireBrick'])); |

## کپی کردن عناصرآرایه با استفاده از () splice

روش بعدی که ما پوشش خواهیم داد ، slice() است. به جای تغییر یک آرایه ، slice() تعداد مشخصی از عناصر را در یک آرایه جدید کپی یا استخراج می کند ، و آرایه ای که با آن فراخوانی شده دست نخورده ترک می‌کند.

slice() تنها 2 پارامتر می‌گیرد - اولی ایندکسی است که از آن می توانید کپی را از آن شروع کنید ، و دوم ایندکسی است که در آن می توان کپی را متوقف کرد (کپی تا این ایندکس اتفاق می افتد ، اما عنصر موجود در این ایندکس را شامل نمی شود). این را در نظر بگیرید:

|  |
| --- |
| let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];  let todaysWeather = weatherConditions.slice(1, 3);  // todaysWeather equals ['snow', 'sleet'];  // weatherConditions still equals ['rain', 'snow', 'sleet', 'hail', 'clear'] |

در واقع ، ما با استخراج عناصر از یک مجموعه موجود ، آرایه جدیدی را ایجاد کرده ایم.

تمرین 5-7: ما یک تابع ، forecast، تعریف کرده ایم که یک آرایه را به عنوان یک آرگومان در نظر می گیرد. تابع را با استفاده از slice() کپی کنید تا اطلاعات را از آرایه آرگومان استخراج کرده و یک آرایه جدید را برگردانید که حاوی عناصر 'warm' و 'sunny' است.

|  |
| --- |
| function forecast(arr) {    return arr;  }  console.log(forecast(['cold', 'rainy', 'warm', 'sunny', 'cool', 'thunderstorms'])); |

## کپی یک آرایه با عملگر spread

در حالی که slice() ما را قادر می سازد عناصر یک آرایه را برای کپی کردن از میان چندین کار مفید دیگر انتخاب کنیم ، عملگر گسترش جدید ES6 به ما این امکان را می دهد تا به راحتی ، با یک نحو ساده و بسیار خواندنی ، همه عناصر یک آرایه را کپی کنیم. نحو گسترش به این سادگی به نظر می رسد: ...

در عمل ، ما می توانیم از عملگرگسترش برای کپی آرایه مانند زیر استفاده کنیم:

|  |
| --- |
| let thisArray = [true, true, undefined, false, null];  let thatArray = [...thisArray];  // thatArray equals [true, true, undefined, false, null]  // thisArray remains unchanged, and is identical to thatArray |

تمرین 5-8: ما یک تابع ، copyMachine تعریف کرده ایم که arr (یک آرایه) و num (یک عدد) را به عنوان آرگومان می‌گیرد. این تابع قرار است آرایه جدیدی را تشکیل دهد که از arr کپی شده به تعداد num ساخته شده است. ما بیشتر کارها را برای شما انجام داده ایم ، اما هنوز درست کار نمی کند. این تابع را با استفاده از نحو spread تغییر دهید تا به درستی کار کند (اشاره: روش دیگری که قبلاً تحت پوشش قرار داده ایم ممکن است در اینجا مفید باشد).

|  |
| --- |
| function copyMachine(arr, num) {    let newArr = [];    while (num >= 1) {      num--;    }    return newArr;  }  console.log(copyMachine([true, false, true], 2)); |

## ترکیب آرایه ها با عملگر spread

یکی دیگر از مزیت های بزرگ عملگر spread ، امکان ترکیب آرایه ها یا قرار دادن تمام عناصر یک آرایه در دیگری با هر ایندکسی است. با نحوهای سنتی تر می توانیم آرایه ها را به هم پیوند دهیم ، اما این تنها به ما امکان می دهد آرایه ها را در انتها و در شروع دیگری ترکیب کنیم. نحو Spread عملیات زیر را بسیار ساده می‌کند:

|  |
| --- |
| let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];  let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander'];  // thatArray now equals ['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander'] |

با استفاده از نحو spread ، ما به تازگی به عملیاتی رسیده ایم كه اگر از رو‌ش‌های سنتی استفاده می كردیم پیچیده تر دارای واژگان زیاد می شد.

تمرین 5-9: ما یک تابع spreadOut تعریف کرده ایم که متغیر sentence را برگرداند. با استفاده از عملگر spread،تابع را تغییر دهید تا آرایه ['learning', 'to', 'code', 'is', 'fun']را برگرداند.

|  |
| --- |
| function spreadOut() {    let fragment = ['to', 'code'];    let sentence; // Change this line    return sentence;  }  console.log(spreadOut()); |

## بررسی وجود داشتن یک عنصر با indexOf()

از آنجا که آرایه ها می توانند در هر زمان تغییر کرده یا جهش پیدا کنند ، هیچ تضمینی در مورد مکان قطعه خاص از داده ها در یک آرایه مشخص وجود ندارد یا حتی اگر این عنصر هنوز وجود داشته باشد.

خوشبختانه ، JavaScript روش داخلی دیگری را با نام indexOf() در اختیار ما قرار می دهد که به ما امکان می دهد به سرعت و به راحتی وجود داشتن یک عنصر را در یک آرایه بررسی کنیم. indexOf() یک عنصر را به عنوان یک پارامتر می‌گیرد و هنگامی که فراخوانی می شود ، موقعیت یا ایندکس آن عنصر را برمی گرداند ، یا اگر عنصر در آرایه وجود نداشته باشد ، 1- را نشان می دهد.

برای مثال:

|  |
| --- |
| let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears'];  fruits.indexOf('dates'); // returns -1  fruits.indexOf('oranges'); // returns 2  fruits.indexOf('pears'); // returns 1, the first index at which the element exists |

تمرین 5-10: indexOf() برای بررسی سریع وجود عنصر در یک آرایه می تواند فوق العاده مفید باشد. ما یک تابع QuickCheck تعریف کرده ایم، که یک آرایه و یک عنصر را به عنوان آرگومان می گیرد. تابع را با استفاده از indexOf() تغییر دهید تا در صورت وجود عنصر ارسال شده در آرایه ، true را برگرداند و در صورت وجود نداشتن false را برگرداند.

|  |
| --- |
| function quickCheck(arr, elem) {    }  console.log(quickCheck(['squash', 'onions', 'shallots'], 'mushrooms')); |

## پیمایش میان همه ی عناصر آرایه با حلقه های for

گاهی اوقات هنگام کار با آرایه ها ، بسیار سخت است که بتوانیم از پیمایش میان هر آیتم ، یک یا چند عنصر مورد نیاز خود را پیدا کنیم ، یا یک آرایه را دستکاری کنیم که براساس آن ، هر داده با معیارهای خاصی مطابقت دارند.

جاوا اسکریپت چندین متد داخلی ساخته شده را ارائه می دهد که هر یک از آنها در آرایه ها به روش های کمی متفاوت برای دستیابی به نتایج متفاوت (مانند every(), forEach(), map()، و غیره) تکرار می شود ، اما تکنیکی که انعطاف پذیر ترین است و بهترین پیشنهاد را برای کنترل مقدار به ما ارائه می دهد یکه حلقه for ساده است.

موارد زیر را در نظر بگیرید:

|  |
| --- |
| function greaterThanTen(arr) {    let newArr = [];    for (let i = 0; i < arr.length; i++) {      if (arr[i] > 10) {        newArr.push(arr[i]);      }    }    return newArr;  }  greaterThanTen([2, 12, 8, 14, 80, 0, 1]);  // returns [12, 14, 80] |

با استفاده از حلقه for ، این تابع از هر عنصر آرایه عبور کرده و به آن دسترسی پیدا می کند و آن را به آزمایش ساده ای که ما ایجاد کرده ایم تسلیم می کند. به این ترتیب ، ما به راحتی تعیین کرده ایم که کدام آیتم های داده از 10 بزرگتر هستند و یک آرایه جدید را که حاوی آن موارد است ، بازگرداندیم.

تمرین 5-11: ما یک تابع ، filteredArray را تعریف کرده ایم که arr ، آرایه ای توخالی و elem را به عنوان آرگومان می گیرد و آرایه جدیدی را برمی گرداند. elem عنصری را نشان می دهد که ممکن است در یک یا بیشتر از آرایه هایی که در داخل arr قرار گرفته اند وجود داشته باشد یا نباشد. تابع را با استفاده از حلقه اصلاح کنید تا یک نسخه فیلتر شده از آرایه ارسال شده برگردانده شود به گونه ای که هر آرایه ای که داخل arr حاوی elem قرار گرفته است حذف شود.

|  |
| --- |
| function filteredArray(arr, elem) {    let newArr = [];    return newArr;  }  console.log(filteredArray([[3, 2, 3], [1, 6, 3], [3, 13, 26], [19, 3, 9]], 3)); |

## آرایه های پیچیده چند بعدی ایجاد کنید

عالی! شما فقط یک تن درباره آرایه ها آموخته اید! این یک بررسی کلی از سطح نسبتاً بالایی بوده است ، و اطلاعات بیشتری در مورد کار با آرایه ها وجود دارد که آن‌ها را در بخش‌های بعدی مشاهده خواهید کرد. اما قبل از این که به دنبال اشیاء برویم ، اجازه دهید نگاهی بیشتر بیندازیم و ببینیم که چگونه آرایه ها می توانند کمی پیچیده تر از آنچه در چالش های قبلی دیده ایم ، شوند.

یکی از قدرتمند ترین ویژگی ها در هنگام فکر کردن به آرایه ها به عنوان ساختار داده ها ، این است که آرایه ها می توانند شامل آرایه های دیگر یا حتی کاملاً از آرایه های دیگر ساخته شده باشند. در چالش های قبلی ما آرایه هایی را دیدیم که حاوی آرایه های دیگر هستند ، اما موارد ساده ای دارند. با این حال ، آرایه ها می توانند شامل عمق نامحدودی از آرایه ها باشند که می توانند شامل آرایه های دیگری باشند ، هر کدام دارای سطح عمق دلخواه خود و غیره هستند. به این ترتیب ، یک آرایه خیلی سریع می تواند به ساختار داده ای بسیار پیچیده تبدیل شود ، که به عنوان یک آرایه چند بعدی یا لانه ای شناخته می شود. مثال زیر را در نظر بگیرید:

|  |
| --- |
| let nestedArray = [ // top, or first level - the outer most array    ['deep'], // an array within an array, 2 levels of depth    [      ['deeper'], ['deeper'] // 2 arrays nested 3 levels deep    ],    [      [        ['deepest'], ['deepest'] // 2 arrays nested 4 levels deep      ],      [        [          ['deepest-est?'] // an array nested 5 levels deep        ]      ]    ]  ]; |

در حالی که ممکن است این مثال پیچیده به نظر برسد ، هنگام برخورد با مقادیر زیادی از داده ها ، این پیچیدگی ناشناخته یا حتی غیرمعمول نیست. با این حال ، ما هنوز هم می توانیم به راحتی به عمیق ترین سطوح یک آرایه این مجموعه با نماد براکت دسترسی داشته باشیم:

|  |
| --- |
| console.log(nestedArray[2][1][0][0][0]);  // logs: deepest-est? |

و اکنون که می دانیم آن قطعه داده در کجا قرار دارد ، در صورت نیاز می توانیم دوباره آن را تنظیم کنیم:

|  |
| --- |
| nestedArray[2][1][0][0][0] = 'deeper still';  console.log(nestedArray[2][1][0][0][0]);  // now logs: deeper still |

تمرین 5-12: ما یک متغیر ، myNestedArray ، برابر با یک آرایه تعریف کرده ایم. myNestedArray را با استفاده از هر ترکیبی از رشته ها ، اعداد و بولین ها برای عناصر داده به گونه ای که دقیقاً دارای پنج سطح عمق است(به یاد داشته باشید که بیرونی ترین آرایه داری سطح 1 است). اصلاح کنید ، در جایی در سطح سوم ،داری رشته 'deep'، در سطح چهارم شامل رشته 'deeper'و در سطح پنجم شامل رشته 'deepest' می شود.

|  |
| --- |
| let myNestedArray = [    ['unshift', false, 1, 2, 3, 'complex', 'nested'],    ['loop', 'shift', 6, 7, 1000, 'method'],    ['concat', false, true, 'spread', 'array'],    ['mutate', 1327.98, 'splice', 'slice', 'push'],    ['iterate', 1.3849, 7, '8.4876', 'arbitrary', 'depth']  ]; |

## افزودن جفت های کلید-مقدار به اشیاء javaScript

در ابتدایی ترین آنها ، اشیاء فقط مجموعه ای از جفت های کلید- مقدار هستند. به عبارت دیگر ، آنها قطعاتی از داده ها (مقادیر) هستند که در شناسه های منحصر به فرد به نام خواص (کلیدها) نگاشت می شوند. به یک مثال نگاهی بیندازید:

|  |
| --- |
| const tekkenCharacter = {    player: 'Hwoarang',    fightingStyle: 'Tae Kwon Doe',    human: true  }; |

کد بالا یک شیء شخصیت بازی ویدیویی Tekken را با نام tekkenCharacter تعریف می کند. آن سه ویژگی دارد که هر کدام از آنها به یک مقدار خاص می پردازند. اگر می خواهید یک ویژگی اضافی مانند "origin"اضافه کنید ، می توان با اختصاص originبه شیء این کار را انجام داد:

|  |
| --- |
| tekkenCharacter.origin = 'South Korea'; |

اگر خاصیت شما فاصله ای داشته باشد یا اگر می خواهید از یک متغیر برای نامگذاری خاصیت استفاده کنید ، نماد براکت مورد نیاز است. در مورد بالا ، این ویژگی با نقل قول ها محصور می شود تا آن را به عنوان یک رشته نشان دهد و دقیقاً مطابق شکل اضافه می شود. بدون نقل قول ، آن به عنوان یک متغیر ارزیابی می‌شود و نام ویژگی هر مقدار که متغیر باشد ، خواهد بود. در اینجا مثالی با متغیر آورده شده است:

|  |
| --- |
| const eyes = 'eye color';  tekkenCharacter[eyes] = 'brown'; |

پس از افزودن همه مثال‌ها ، این شیء به این صورت خواهد بود:

|  |
| --- |
| {    player: 'Hwoarang',    fightingStyle: 'Tae Kwon Doe',    human: true,    origin: 'South Korea',    'hair color': 'dyed orange',    'eye color': 'brown'  }; |

تمرین 5-13: یک شی foods با سه ورودی ایجاد شده است. با استفاده از نحو مورد نظر خود ، سه ورودی دیگر به آن اضافه کنید: bananas با مقدار 13 ، grapes با مقدار 35 و strawberries با مقدار 27.

|  |
| --- |
| let foods = {    apples: 25,    oranges: 32,    plums: 28  };  console.log(foods); |

## شی موجود درون شی دیگر را تعقیر دهید

حال اجازه دهید نگاهی به یک موضوع کمی پیچیده تر بیندازیم. خواص شی را می توان درون عمق دلخواه قرار داد و مقادیر آنها می تواند هر نوع داده ای باشد که توسط JavaScript پشتیبانی می شود ، از جمله آرایه ها و حتی اشیاء دیگر. موارد زیر را در نظر بگیرید:

|  |
| --- |
| let nestedObject = {    id: 28802695164,    date: 'December 31, 2016',    data: {      totalUsers: 99,      online: 80,      onlineStatus: {        active: 67,        away: 13,        busy: 8      }    }  }; |

شیئ تودرتو دارای سه ویژگی است: id (مقدار یک عدد است) ، تاریخ (مقدار یک رشته است) و داده ها (مقدار یک شی با ساختار تو در تو است). در حالی که ساختارها می توانند به سرعت پیچیده شوند ، ما هنوز می توانیم از همان نمادها برای دسترسی به اطلاعات مورد نیاز استفاده کنیم. برای اختصاص مقدار 10 به ویژگی busy، onlineStatus در حالت تو در تو ، از مرجع نقطه برای مرجع خاصیت استفاده می کنیم:

|  |
| --- |
| nestedObject.data.onlineStatus.busy = 10; |

تمرین 5-14: در اینجا ما یک شی userActivity تعریف کرده ایم ، که شامل یک شی دیگر است که درون آن قرار دارد. مقدار کلید onlin را روی 45 تنظیم کنید.

|  |
| --- |
| let userActivity = {    id: 23894201352,    date: 'January 1, 2017',    data: {      totalUsers: 51,      online: 42    }  };  console.log(userActivity); |

## دسترسی به نام های خاصیت با علامت براکت

در اولین چالش شیء ، ما از استفاده از نماد براکت به عنوان راهی برای دستیابی به مقادیر خاصیت با استفاده از ارزیابی یک متغیر نام بردیم. به عنوان مثال ، تصور کنید که از شیء foods ما در برنامه ای برای ثبت نام پول نقد سوپر مارکت استفاده می شود.

ما برخی توابع را داریم که selectedFoodرا تعیین می کنند و می خواهیم شی foods خود را از نظر وجود آن مواد غذایی بررسی کنیم. این ممکن است اینگونه به نظر برسد:

|  |
| --- |
| let selectedFood = getCurrentFood(scannedItem);  let inventory = foods[selectedFood]; |

این کد مقدار ذخیره شده در متغیر selectedFood را ارزیابی کرده و مقدار آن کلید را در شی foods یا در صورت وجود نداشتن آن undefined ، تعریف می کند. نماد براکت بسیار مفید است زیرا بعضی اوقات ویژگی های اشیاء قبل از زمان اجرا مشخص نیستند یا ما باید به روش پویاتر به آنها دسترسی پیدا کنیم.

تمرین 5-15: ما یک تابع ، checkInventory تعریف کرده ایم ، که یک آیتم اسکن شده را به عنوان یک آرگومان دریافت می کند. مقدار فعلی کلید scannedItem را در شی foods برگردانید. می توانید فرض کنید که فقط کلیدهای معتبر به عنوان آرگومان برای checkInventory ارائه خواهند شد.

|  |
| --- |
| let foods = {    apples: 25,    oranges: 32,    plums: 28,    bananas: 13,    grapes: 35,    strawberries: 27  };  function checkInventory(scannedItem) {  }  console.log(checkInventory("apples")); |

## از کلیدواژه delete برای حذف خواص اشیاء استفاده کنید

اکنون می دانید که اشیاء و خصوصیات اصلی و مزایای آنها چیست. به طور خلاصه ، آنها ذخیره کننده کلید- مقدار هستند که روشی انعطاف پذیر و شهودی برای ساختار داده ها ارائه می دهند و زمان جستجوی خیلی سریع را ارائه می دهند. در طول بقیه این چالش ها ، ما چندین عملیات معمول را که می توانید روی اشیاء انجام دهید ، شرح خواهیم داد تا بتوانید استفاده از این ساختارهای داده مفید در برنامه های تان راحت شود.

در چالش های قبلی ، ما هر دو جفت کلید- مقدار یک شی را اضافه کرده ایم و اصلاح کرده ایم. در اینجا خواهیم دید که چگونه می توانیم یک جفت کلید- مقدار را از یک شی حذف کنیم.

بیایید بار دیگر به عنوان مثال foods خود را برای بار آخر بازنگری کنیم. اگر می خواستیم کلید apples را حذف کنیم ، می توانیم آن را با استفاده از کلید واژه delete مانند این حذف کنیم:

|  |
| --- |
|  |

تمرین 5-16: با استفاده از کلید واژه delete ، کلیدهای oranges، plums و strawberries را از شی foods حذف کنید.

|  |
| --- |
| let foods = {    apples: 25,    oranges: 32,    plums: 28,    bananas: 13,    grapes: 35,    strawberries: 27  };  console.log(foods); |

## بررسی کنید که آیا یک شیء دارای خاصیت است

اکنون می توانیم کلیدها را به اشیاء اضافه کنیم ، تغییر دهیم و حذف کنیم. اما اگر فقط بخواهیم بدانیم آیا یک شی خاصیت خاصی دارد یا خیر ، چه می شود؟ جاوا اسکریپت دو روش مختلف برای این کار در اختیار ما قرار می دهد. اولی از متد hasOwnProperty()استفاده می کند و دیگری از کلمه کلیدی in استفاده می کند. اگر ما یک شیء users با خاصیت Alan داریم ، می توانیم از یکی از روش های زیر وجود داشتن آن را بررسی کنیم:

|  |
| --- |
| users.hasOwnProperty('Alan');  'Alan' in users;  // both return true |

تمرین 5-17: ما با استفاده از برخی از کاربران یک شیء users ایجاد کرده ایم و یک تابع isEveryoneHere است که ما users را به عنوان یک آرگومان به آن ارسال می کنیم. نوشتن این تابع را به پایان برسانید تا فقط اگر شیء users شامل چهار نام ، Alan ، Jeff ، Sarah و Ryan به عنوان کلید باشد،true ، و در غیر این صورت false را برگرداند.

|  |
| --- |
| let users = {    Alan: {      age: 27,      online: true    },    Jeff: {      age: 32,      online: true    },    Sarah: {      age: 48,      online: true    },    Ryan: {      age: 19,      online: true    }  };  function isEveryoneHere(obj) {  }  console.log(isEveryoneHere(users)); |

## پیمایش میان کلید های یک شی با یک دستور for … in

بعضی اوقات ممکن است لازم باشد که تمام کلیدهای موجود در یک شی را پیمایش کنید. این به یک نحو خاص در جاوا اسکریپت احتیاج دارد که دستور for...in نامیده می‌شود.برای شی users ما ، این می تواند مانند زیر باشد:

|  |
| --- |
| for (let user in users) {    console.log(user);  }  // logs:  Alan  Jeff  Sarah  Ryan |

در این دستور ، ما یک متغیر userرا تعریف کردیم و همانطور که مشاهده می کنید ، این متغیر در طول هر پیمایش به هر یک از کلیدهای شی به عنوان دستور در بین شی چرخیده، و در نتیجه نام هر کاربر روی کنسول چاپ می شود.

توجه: اشیاء ترتیب كلیدهای ذخیره شده را مانند آرایه ها حفظ نمی كنند. بنابراین موقعیت کلید بر روی یک شی یا ترتیب نسبی که در آن ظاهر می شود ، هنگام مراجعه یا دستیابی به آن کلید بی ربط نیست.

تمرین 5-18: از دستور for...in درون این تابع استفاده کنید تا میان شیء کاربرانی که به این تابع ارسال شده اند بچرخید و تعداد کاربرانی را که خاصیت online آنها true است را برگردانید. مثالی از یک شی کاربران که می توانند به countOnline ارسال شوند ، در زیر نشان داده شده است. هر کاربر یک خاصیت online با مقدار true یا false خواهد داشت..

|  |
| --- |
| {    Alan: {      online: false    },    Jeff: {      online: true    },    Sarah: {      online: false    }  } |

|  |
| --- |
| function countOnline(usersObj) {    //فقط این قسمت را تعقیر دهید  } |

## آرایه ای از همه کلید های شی را با () Object.keys ایجاد کنید

همچنین می توانیم با استفاده از متد () Object.keys ، آرایه ای تولید کنیم که شامل تمام کلیدهای ذخیره شده در یک شی است و آن را در یک شی به عنوان آرگومان ارسال کنیم. این یک آرایه با رشته هایی را نشان می‌دهد که هر خاصیت موجود در شی را نشان می دهد. باز هم ، سفارش خاصی برای ورودی های موجود در آرایه وجود نخواهد داشت.

تمرین 5-18: نوشتن تابع getArrayOfUsers را تمام کنید تا یک آرایه حاوی تمام خصوصیات موجود در شیء دریافت شده به عنوان آرگومان را برگرداند.

|  |
| --- |
| let users = {    Alan: {      age: 27,      online: false    },    Jeff: {      age: 32,      online: true    },    Sarah: {      age: 48,      online: false    },    Ryan: {      age: 19,      online: true    }  };  function getArrayOfUsers(obj) {    }  console.log(getArrayOfUsers(users)); |

## آرایه ذخیره شده در یک شی را اصلاح کنید

اکنون همه کارهای اساسی را برای اشیاء JavaScript مشاهده کرده اید. می توانید جفت های کلید-مقدار را اضافه کنید ، تغییر دهید و حذف کنید ، بررسی کنید که اگر کلید وجود داشته باشد و بیش از همه کلیدهای یک شیء تکرار شود. با یادگیری JavaScript ، برنامه های متنوع تری از اشیاء مشاهده خواهید کرد.

علاوه بر این ، درس ساختار داده های واقع در بخش برنامه نویسی ، همچنین ES6 Map و تنظیم اشیاء را ، که هر دو شبیه به اشیاء معمولی هستند اما برخی از ویژگی های اضافی را ارائه می دهند پوشش می دهد. اکنون که اصول اولیه آرایه ها و اشیاء را آموخته اید ، کاملاً آماده هستید تا با استفاده از JavaScript برای مقابله با مشکلات پیچیده تر بروید!

تمرین 5-20: به شیء مورد نظر در ویرایشگر کد نگاهی بیندازید. شی user شامل سه کلید است. کلید data شامل پنج کلید است که یکی از آنها حاوی آرایه ای از friends است. از این رو می توانید ببینید که اشیاء به عنوان ساختار داده چقدر انعطاف پذیر هستند. ما شروع به نوشتن یک تابع addFriend کرده ایم. نوشتن آن را به پایان برسانید تا یک شی user را بگیرد و نام آرگومان friend را به آرایه ذخیره شده در user.data.friends اضافه کرده و آن آرایه را بازگرداند.

|  |
| --- |
| let user = {    name: 'Kenneth',    age: 28,    data: {      username: 'kennethCodesAllDay',      joinDate: 'March 26, 2016',      organization: 'freeCodeCamp',      friends: [        'Sam',        'Kira',        'Tomo'      ],      location: {        city: 'San Francisco',        state: 'CA',        country: 'USA'      }    }  };  function addFriend(userObj, friend) {  }  console.log(addFriend(user, 'Pete')); |

# 6. الگوریتم پایه اسکریپت نویسی

## آشنایی با الگوریتم پایه اسکریپت نویسی

الگوریتم رایانه دنباله ای از مراحل است که برای دستیابی به یک نتیجه خاص دنبال می شود. برای نوشتن یک الگوریتم ، ابتدا باید یک مشکل را درک کنید ، و سپس آن را با کد نویسی حل کنید.

برای آسان تر کردن حل مشکلات ، تجزیه آنها در بخش های زیادی مفید خواهد بود. سپس ، هر قطعه می تواند یک به یک حل شود. به عنوان مثال ، اگر در حال ساخت ماشین حساب هستید ، سعی نکنید مشکل را به طور کلی حل کنید. ابتدا نحوه دریافت ورودی ها را در نظر بگیرید. سپس هر عمل حسابی را یکی یکی مشخص کنید. در آخر ، نتایج را نمایش دهید.

در این بخش یاد می گیریم مشکلات الگوریتمی پایه را با استفاده از JavaScript حل کنیم. این به شما در بهبود مهارت های حل مسئله کمک می‌‎کند و شما را برای حل مشکلات پیچیده تر بعداً آماده می کند.

نکته: اگر گیر افتادید ، از console.log() استفاده کنید تا مقادیر متغیر را در کنسول نشان دهید. این به اشکال زدایی مشکلات کمک می کند.

## تمرین 6-1: تبدیل سلسیوس به فارنهایت

الگوریتم برای تبدیل از سلسیوس به فارنهایت درجه حرارت در زمان های سلسیوس 9/5 به علاوه 32 است.

به شما یک متغیر celsius داده می شود که نشان دهنده درجه حرارت در سلسیوس است. از متغیر fahrenheit که در حال حاضر تعریف شده استفاده کنید و به به دمای فارنهایت تخصیصش دهید و معادل آن را بدست آورید. برای کمک به تبدیل دمای سلسیوس به فارنهایت از الگوریتم ذکر شده در بالا استفاده کنید.

|  |
| --- |
| function convertToF(celsius) {    let fahrenheit;    return fahrenheit;  }  convertToF(30); |

## تمرین 6-2: معکوس یک رشته

رشته ارائه شده را معکوس کنید.

ممکن است لازم باشد رشته را به یک آرایه تبدیل کنید تا بتوانید آنرا معکوس کنید.

نتیجه شما باید رشته باشد.

|  |
| --- |
| function reverseString(str) {    return str;  }  reverseString("hello"); |

## تمرین 6-3: فاکتورگیری یک عدد

فاکتوریل عدد صحیح ارائه شده را برگردانید.

اگر عدد صحیح با حرف n نشان داده شود ، یک فاکتوریل حاصلی است که تمام اعداد صحیح مثبت کمتر یا برابر با n باشد.

فاکتوریل ها معمولاً با علامت تعجب نشان داده می شوند n!

برای مثال : 5! = 1 \* 2 \* 3 \* 4 \* 5 = 120

فقط اعداد صحیح بزرگتر یا مساوی صفر به این تابع ارسال می شود.

|  |
| --- |
| function factorialize(num) {    return num;  }  factorialize(5); |

## تمرین 6-4: طولانی ترین کلمه را در یک رشته پیدا کنید

طول طولانی ترین کلمه را در جمله ارائه شده برگردانید.

پاسخ شما باید یک عدد باشد

|  |
| --- |
| function findLongestWordLength(str) {    return str.length;  }  findLongestWordLength("The quick brown fox jumped over the lazy dog") |

## تمرین 6-5: بازگرداندن بزرگترین اعداد در آرایه

آرایه ای متشکل از بزرگترین عدد را از هر زیر آرایه ارائه دهید. برای سادگی ، آرایه ارائه شده دقیقاً شامل 4 زیر آرایه است.

به یاد داشته باشید ، می توانید از طریق یک یک حلقه میان عناصر آرایه حرکت کنید و با نحو arr[i] به هر عضو دسترسی پیدا کنید.

|  |
| --- |
| function largestOfFour(arr) {    return arr;  }  largestOfFour([[4, 5, 1, 3], [13, 27, 18, 26], [32, 35, 37, 39], [1000, 1001, 857, 1]]); |

## تمرین 6-6: پایان را تأیید کنید

بررسی کنید که آیا یک رشته (آرگومان اول ، str) با رشته هدف مشخص شده به پایان می رسد (آرگومان دوم ، target).

این چالش را می توان با متد endsWith()که در ES2015 معرفی شده است ، حل کرد. اما برای هدف این چالش ، ما دوست داریم که به جای آن از یکی از متد های بستر JavaScript استفاده کنید.

|  |
| --- |
| function confirmEnding(str, target) {    return str;  }  confirmEnding("Bastian", "n"); |

## تمرین 6-7: یک رشته را تکرار کنید یک رشته را تکرار کنید

یک رشته داده شده str (آرگومان اول) را به تعداد num تکرار کنید (آرگومان دوم). اگر num مثبت نیست یک رشته خالی برگردانید.

|  |
| --- |
| function repeatStringNumTimes(str, num) {    return str;  }  repeatStringNumTimes("abc", 3); |

## تمرین 6-8: یک رشته را کوتاه کنید

اگر رشته داده شده (آرگومان اول) طولانی تر از حداکثر طول رشته داده شده باشد (آرگومان دوم) رشته را کوتاه کنید. رشته کوتاه را با یک ... در انتهایش برگردانید.

|  |
| --- |
| function truncateString(str, num) {    return str;  }  truncateString("A-tisket a-tasket A green and yellow basket", 8); |

## تمرین 6-9: پیداکننده نگهدارنده ها

تابعی را ایجاد کنید که در میان یک آرایه arr جستجو کند و اولین عنصر موجود در آن را که از عبور 'truth test' می‌کند را برگرداند. این بدان معناست که با توجه به عنصر x ، 'truth test' ارسال می شود اگر func(x) ، true باشد. اگر هیچ عنصری از آزمون عبور نمی کند ، undefined را بازگردانید.

|  |
| --- |
| function findElement(arr, func) {    let num = 0;    return num;  }  findElement([1, 2, 3, 4], num => num % 2 === 0); |

## تمرین 6-10: Boo who

بررسی کنید که آیا یک مقدار به عنوان بولین اولیه طبقه بندی شده است یا خیر. true یا false را برگردانید.

بولین های اولیه true و false هستند.

|  |
| --- |
| function findElement(arr, func) {    let num = 0;    return num;  }  findElement([1, 2, 3, 4], num => num % 2 === 0); |

## تمرین 6-11: حرف اول جمله

رشته اوليه را با حرف اول هر كلمه بزرگ برگردانيد. اطمینان حاصل کنید که بقیه کلمه دارای حروف کوچک است. برای اهداف این تمرین ، شما همچنین باید کلمات متصل کننده مانند "the" و "of" را به به حروف بزرگ تبدیل کنید.

|  |
| --- |
| function titleCase(str) {    return str;  }  titleCase("I'm a little tea pot"); |

## تمرین 6-12: Slice و Splice

به شما دو آرایه و یک ایندکس داده می شود. به ترتیب هر عنصر از آرایه اول را در آرایه دوم کپی کنید.

درج عناصراز ایندکس n در آرایه دوم شروع کنید. آرایه نتیجه را برگردانید. آرایه های ورودی پس از اجرای تابع باید یکسان باشند.

|  |
| --- |
| function frankenSplice(arr1, arr2, n) {    return arr2;  }  frankenSplice([1, 2, 3], [4, 5, 6], 1); |

## تمرین 6-13: Falsy Bouncer

مام مقادیر غلط را از یک آرایه جدا کنید.

مقادیر کاذب در JavaScript false ، null ، 0، "" ، undefinedو NaN است.

نکته: سعی کنید هر مقدار را به یک بولین تبدیل کنید.

|  |
| --- |
| function bouncer(arr) {    return arr;  }  bouncer([7, "ate", "", false, 9]); |

## تمرین 6-14: من به کجا تعلق دارم

کمترین ایندکسی را که در آن مقدار (آرگومان دوم) باید پس از آنکه مرتب شد ، وارد آرایه شود برگردانید. مقدار برگشتی باید یک عدد باشد.

به عنوان مثال ، getIndexToIns([1,2,3,4], 1.5) باید 1 را برگرداند زیرا از 1 بزرگتر است (شاخص 0) ، اما کمتر از 2 (ایندکس 1) است.

به همین ترتیب ، getIndexToIns([20,3,5], 19) باید 2را برگرداند زیرا پس از مرتب سازی آرایه ، آن اینگونه به نظر می رسد [3,5,20] و 19 کمتر از 20 (ایندکس 2) و بیشتر از 5 (ایندکس 1) است.

|  |
| --- |
| function getIndexToIns(arr, num) {    return num;  }  getIndexToIns([40, 60], 50); |

## تمرین 6-15: Mutations

اگر رشته موجود در اولین عنصر آرایه شامل تمام حروف رشته در عنصر دوم آرایه باشد ، true را برگردانید. به عنوان مثال ، ["hello", "Hello"]باید true برگردد زیرا تمام حروف در رشته دوم وجود دارد ، بدون توجه به بزرگ یا کوچک بودن حروف در رشته اول .

آرگومان های ["hello", "hey"] باید false را برگرداند زیرا رشته "hello" حاوی "y" نیست. سرانجام ، ["Alien", "line"]، باید true برگرداند زیرا تمام حروف موجود در "line" در "Alien" وجود دارد.

|  |
| --- |
| function mutation(arr) {    return arr;  }  mutation(["hello", "hey"]); |

## تمرین 6-16: Chunky Monkey

تابعی را بنویسید که یک آرایه (آرگومان اول) را به اندازه طول (آرگومان دوم) به گروهی split کند و آنها را به صورت آرایه ای دو بعدی برگرداند.

|  |
| --- |
| function chunkArrayInGroups(arr, size) {    return arr;  }  chunkArrayInGroups(["a", "b", "c", "d"], 2); |

# 7. برنامه نویسی شی گرا

## مقدمه ای بر چالش های برنامه نویسی شی گرا

در هسته اصلی خود ، توسعه نرم افزار مشکلی را حل می کند یا با محاسبه نتیجه می گیرد. فرآیند توسعه نرم‌افزار ابتدا یک مشکل را تعریف می کند ، سپس یک راه حل ارائه می دهد. برنامه نویسی شی گرا یکی از چندین رویکرد مهم در روند توسعه نرم افزار است.

همانطور که از نام آن پیداست ، برنامه نویسی شی گرا کد را در تعاریف شی سازمان می دهد. این ها بعضی اوقات کلاس نامیده می شوند ، و آنها داده ها را با رفتار مرتبط گروه بندی می کنند. داده ها ویژگی های شی هستند و رفتار (یا توابع) متد ها هستند.

ساختار شی آن را در داخل یک برنامه انعطاف پذیر می کند. اشیا می توانند با فراخوانی و انتقال داده ها به متدهای شی دیگر ، اطلاعات را انتقال دهند. همچنین ، کلاس های جدید می توانند کلیه ویژگی ها را از یک کلاس پایه یا والد دریافت یا به ارث ببرند. این به کاهش کد تکراری کمک می کند.

انتخاب رویکرد برنامه نویسی شما به چند عامل بستگی دارد. این موارد شامل نوع مسئله و همچنین چگونگی ساختاردهی داده ها و الگوریتم های شما می باشد. این بخش اصول برنامه نویسی شی گرا را در JavaScript پوشش می دهد.

## ایجاد یک شی پایه JavaScript

درباره چیزهایی فکر کنید که مردم هر روز می بینند ، مانند ماشین ها ، مغازه ها و پرندگان. اینها همه اشیا هستند: چیزهای ملموسی که مردم می توانند مشاهده کنند و با آنها تعامل داشته باشند.

برخی از خصوصیات این اشیا چیست؟ یک ماشین چرخ دارد. مغازه ها اقلام می فروشند. پرندگان بال دارند. این خصوصیات ، مشخص کننده آن است که چه چیزی یک شی را تشکیل می دهد. توجه داشته باشید که اشیا مشابه خصوصیات یکسانی دارند ، اما ممکن است مقادیر مختلفی برای آن خصوصیات داشته باشند. به عنوان مثال ، همه اتومبیل ها دارای چرخ هستند ، اما همه اتومبیل ها دارای تعداد چرخ یکسان نیستند.

اشیا در JavaScript برای مدل سازی اشیا-در دنیای واقعی استفاده می شوند ، دقیقاً مانند همتایان دنیای واقعی به آنها ویژگی و رفتار می دهند. در اینجا مثالی با استفاده از این مفاهیم برای ایجاد یک شی اردک آورده شده است:

|  |
| --- |
| let duck = {    name: "Aflac",    numLegs: 2  }; |

این شی اردک دارای دو جفت وپژگی-مقدار است: یکی name، "Aflac" و یک عدد numLegs، 2.

تمرین 7-1: یک شی dog با مشخصات name و numLegs ایجاد کنید و آنها را به ترتیب روی یک رشته و یک عدد تنظیم کنید.

|  |
| --- |
| let dog = {}; |

## برای دسترسی به خصوصیات یک شی از نقطه گذاری استفاده کنید

آخرین چالش یک شی object را با خواص مختلف ایجاد کرد. اکنون نحوه دستیابی به مقادیر آن ویژگی ها را خواهید دید. مثالی در اینجا آورده شده است:

|  |
| --- |
| let duck = {    name: "Aflac",    numLegs: 2  };  console.log(duck.name);  // This prints "Aflac" to the console |

برای دستیابی به مقدار "Aflac" از علامت نقطه بر روی نام شیduck و به دنبال آن نام خاصیت ،استفاده می‌شود.

تمرین 7-2: هر دو ویژگی شی dogرا در کنسول خود چاپ کنید.

|  |
| --- |
| let dog = {    name: "Spot",    numLegs: 4  }; |

## یک متد بر روی یک شی ایجاد کنید

اشیا می توانند نوع خاصی از خاصیت داشته باشند که متد نامیده می شود. متد ها خصوصیاتی هستند که توابع هستند. این رفتار متفاوت را به یک شی اضافه می کند. در اینجا مثال duck با یک روش آورده شده است:

|  |
| --- |
| let duck = {    name: "Aflac",    numLegs: 2,    sayName: function() {return "The name of this duck is " + duck.name + ".";}  };  duck.sayName();  // Returns "The name of this duck is Aflac." |

مثال متد sayName را اضافه می کند که تابعی است که جمله ای را با نام duck برمی گرداند. توجه داشته باشید که این متد با استفاده از duck.name به ویژگی name در دستور return دسترسی پیدا کرده است. چالش بعدی روش دیگری برای انجام این کار را پوشش می دهد.

تمرین 7-3: با استفاده از شی dog ، متدی به نام sayLegs به آن بدهید. متد باید جمله "This dog has 4 legs." را برگرداند.

|  |
| --- |
| let dog = {    name: "Spot",    numLegs: 4,  };  dog.sayLegs(); |

## با استفاده ازکلمه کلیدی this کد را بیشتر قابل استفاده کنید

آخرین چالش متدی را به شی douk وارد کرد. آن از نماد نقطه duck.name برای دستیابی به مقدار خاصیت name در دستور return استفاده کرده است:

sayName: function() {return "The name of this duck is " + duck.name + ".";}

در حالی که این یک روش معتبر برای دسترسی به خاصیت شی است ، اما در اینجا یک دام وجود دارد. اگر نام متغیر تغییر کند ، هر کدی که به نام اصلی ارجاع می کند نیز باید به روز شود. در یک تعریف کوتاه شی ، آن مشکلی ندارد ، اما اگر یک شی ارجاعات زیادی به خصوصیات اش داشته باشد ، احتمال خطای بیشتری وجود دارد.

راهی برای جلوگیری از این مسائل با استفاده از کلمه کلیدی this است:

|  |
| --- |
| let duck = {    name: "Aflac",    numLegs: 2,    sayName: function() {return "The name of this duck is " +  this.name + ".";}  } |

این یک موضوع عمیق است و مثال فوق تنها یکی از روش‌های استفاده از آن است. در زمینه فعلی ، this به شیئی اشاره دارد که متد با آن مرتبط است: douk. در صورت تغییر نام شی به mallard ، نیازی به یافتن تمام ارجاعات به douk در کد نیست. این کد را قابل استفاده مجدد و آسان تر برای خواندن می کند.

تمرین 7-4: متد dog.sayLegs را اصلاح کنید تا هرگونه ارجاع به dog حذف شود. برای راهنمایی از مثال duck استفاده کنید.

|  |
| --- |
| let dog = {    name: "Spot",    numLegs: 4,    sayLegs: function() {return "This dog has " + dog.numLegs + " legs.";}  };  dog.sayLegs(); |

## یک تابع سازنده را تعریف کنید

سازنده ها توابعی هستند که اشیا جدید ایجاد می کنند. آنها خصوصیات و رفتارهایی را تعریف می کنند که به شی جدید تعلق داشته باشند. آنها را به عنوان طرحی برای خلق اشیا جدید تصور کنید.

## در اینجا مثالی از سازنده آورده شده است:

|  |
| --- |
| function Bird() {    this.name = "Albert";    this.color = "blue";    this.numLegs = 2;  } |

این سازنده یک شی Bird را به ترتیب با مشخصه های name ، color و numLegs روی Albert ، blue و 2 تعریف می کند. سازندگان از چند قرارداد را پیروی می کنند:

* سازنده ها با یک نام حروف بزرگ تعریف می شوند تا آنها را از تابع های دیگر که سازنده نیستند متمایز کند.
* سازندگان از کلمه کلیدی this برای تعیین خصوصیات شی ایجاد شده استفاده می کنند. در داخل سازنده ، this به شی جدیدی که ایجاد خواهد کرد اشاره دارد.
* سازندگان به جای بازگرداندن مقداری ، همانند سایر توابع ، خصوصیات و رفتارها را تعریف می کنند.

تمرین 7-5: یک سازنده ، Dog ، با مشخصات ، name، color و numLegs ایجاد کنید که به ترتیب روی یک رشته ، یک رشته و یک شماره تنظیم شده اند.

## برای ایجاد اشیا از سازنده استفاده کنید

در اینجا سازنده Bird از چالش قبلی آمده است:

|  |
| --- |
| function Bird() {    this.name = "Albert";    this.color  = "blue";    this.numLegs = 2;    // "this" inside the constructor always refers to the object being created  }  let blueBird = new Bird(); |

توجه داشته باشید که هنگام تماس با سازنده از عملگر new استفاده می شود. this به JavaScript می گوید تا نمونه جدیدی از Bird به نام blueBird ایجاد کند. بدون عملگر new ،this در داخل سازنده به شی تازه ایجاد شده اشاره نمی کند و نتایج غیر منتظره ای به بار می آورد. اکنون blueBird دارای تمام خصوصیاتی است که در داخل سازنده Bird تعریف شده است:

blueBird.name; // => Albert

blueBird.color; // => blue

blueBird.numLegs; // => 2

دقیقاً مانند هر شی دیگر ، می توان به خصوصیات آن دسترسی پیدا کرد و اصلاح کرد:

blueBird.name = 'Elvira';

blueBird.name; // => Elvira

تمرین 7-6: از سازنده Dog در آخرین درس برای ایجاد نمونه جدیدی از Dog ، با اختصاص دادن آن به یک متغیر houndاستفاده کنید.

|  |
| --- |
| function Dog() {    this.name = "Rupert";    this.color = "brown";    this.numLegs = 4;  }  // Only change code below this line  function Bird() {    this.name = "Albert";    this.color  = "blue";    this.numLegs = 2;    // "this" inside the constructor always refers to the object being created  }  let blueBird = new Bird(); |

## سازنده‌ها را برای دریافت آرگومان ها گسترش دهید

سازندگان Bird و Dog از آخرین چالش به خوبی کار کردند. با این حال ، توجه داشته باشید که تمام پرنده‌هایی که با سازنده Bird ایجاد می شوند ، به طور خودکار Albert نامیده می شوند ، رنگ آبی و دو پا دارند. اگر پرنده هایی با مقادیر مختلف برای نام و رنگ می خواهید ، چه می کنید؟ تغییر خصوصیات هر پرنده به صورت دستی امکان پذیر است اما کار بسیار زیادی خواهد بود:

|  |
| --- |
| let swan = new Bird();  swan.name = "Carlos";  swan.color = "white"; |

فرض کنید شما در حال نوشتن برنامه ای برای ردیابی صدها یا حتی هزاران پرنده مختلف در یک مرغداری بودید. برای ایجاد همه پرندگان زمان زیادی لازم است و سپس خصوصیات با مقادیر مختلف برای هر یک تغییر می یابد. برای ایجاد آسان تر اشیا مختلف Bird ، می توانید سازنده Bird خود را طوری طراحی کنید که پارامترها را بپذیرد:

|  |
| --- |
| function Bird(name, color) {    this.name = name;    this.color = color;    this.numLegs = 2;  } |

سپس مقادیر را به عنوان آرگومان برای تعریف هر پرنده منحصر به فرد به سازنده Bird وارد کنید: let cardinal = new Bird("Bruce", "red"); این مورد نمونه جدیدی از Bird را با مشخصات نام و رنگ به ترتیب Bruce و red تنظیم می کند. ویژگی numLegs هنوز روی 2 تنظیم شده است. cardinal دارای این خواص است:

|  |
| --- |
| cardinal.name // => Bruce  cardinal.color // => red  cardinal.numLegs // => 2 |

سازنده انعطاف پذیرتر است. اکنون می توان خصوصیات مربوط به هر Bird را در زمان ایجاد آن تعریف کرد ، این یکی از روش‌های بسیار مفید سازندگان جاوا اسکریپت است.

آنها اشیا را براساس ویژگی‌ها و رفتار مشترک با هم گروه بندی می کنند و نقشه ای را تعریف می کنند که ایجاد آنها را خودکار می کند.

تمرین 7-7: سازنده Dog دیگری ایجاد کنید. این بار آنرا تنظیم کنید تا name و color پارامترها را بگیرد و ویژگی numLegs را در 4 ثابت کند. سپس یکDog جدید ذخیره شده در متفیر terrierایجاد کنید. به آن دو رشته به عنوان آرگومان برای خصوصیات nameو colorارسال کنید.

|  |
| --- |
| function Dog() {} |

## سازنده سازنده شی را با instanceof تایید کنید

هر زمان یک تابع سازنده یک شی جدید ایجاد می کند ، گفته می شود که آن شی نمونه ای از سازنده آن است. جاوا اسکریپت یک روش راحت برای تأیید این مورد با عملگر instanceofارائه می دهد. Instanceof به شما امکان مقایسه یک شی با سازنده را می دهد ، true یا false را بر اساس اینکه آیا آن شی با سازنده ایجاد شده است یا خیر برمی‌گرداند. مثالی در اینجا آورده شده است:

|  |
| --- |
| let Bird = function(name, color) {    this.name = name;    this.color = color;    this.numLegs = 2;  }  let crow = new Bird("Alexis", "black");  crow instanceof Bird; // => true |

اگر یک شی بدون استفاده از سازنده ایجاد شود ، instanceofتأیید می کند که آن نمونه ای از آن سازنده نیست:

|  |
| --- |
| let canary = {    name: "Mildred",    color: "Yellow",    numLegs: 2  };  canary instanceof Bird; // => false |

تمرین 7-8: نمونه جدیدی از سازنده House ایجاد کنید ، آن را myHouse بنامید و یک عدد به عنوان تعداد اتاق خواب ها به آن ارسال کنید. سپس ، با استفاده از instanceof تأیید کنید که این یک نمونه از House است.

|  |
| --- |
| function House(numBedrooms) {    this.numBedrooms = numBedrooms;  } |

## خواص اختصاصی را درک کنید

در مثال زیر ، سازنده Bird دو ویژگی را تعریف می کند: name و numLegs:

|  |
| --- |
| function Bird(name) {    this.name  = name;    this.numLegs = 2;  }  let duck = new Bird("Donald");  let canary = new Bird("Tweety"); |

name و numLegs خواص اختصاصی نامیده می شوند ، زیرا آنها مستقیماً روی شی نمونه تعریف می‌شوند. این بدان معناست که duck و canary هر یک نسخه جداگانه از این خواص را دارند. در واقع هر نمونه از Bird نسخه خاص خود را از این خواص دارند. کد زیر تمام خواص اختصاصی ، duck را به آرایه ownProps اضافه می‌کند: سازنده دو ویژگی را تعریف می کند: name و numLegs:

|  |
| --- |
| let ownProps = [];  for (let property in duck) {    if(duck.hasOwnProperty(property)) {      ownProps.push(property);    }  }  console.log(ownProps); // prints [ "name", "numLegs" ] |

تمرین 7-9: خواص اختصاصی canary را به آرایه ownProps اضافه کنید.

|  |
| --- |
| function Bird(name) {    this.name = name;    this.numLegs = 2;  }  let canary = new Bird("Tweety");  let ownProps = []; |

## از خواص نمونه اولیه برای کاهش کد تکراری استفاده کنید

از آنجا که numLegs احتمالاً برای همه نمونه های Bird مقدار یکسانی دارد ، در واقع در هر نمونه Bird متغیر numLegs تکراری دارید.

این ممکن است مسئله ای نباشد وقتی که فقط دو نمونه وجود دارد ، اما تصور کنید اگر میلیون ها نمونه وجود داشته باشد. این می تواند متغیرهای تکراری زیادی باشد.

یک روش بهتر استفاده از نمونه اولیه Bird است. خصوصیات موجود در نمونه اولیه در میان همه نمونه های پرنده به اشتراک گذاشته شده است. در اینجا نحوه افزودن numLegs به نمونه اولیه Bird آورده شده است:

|  |
| --- |
| Bird.prototype.numLegs = 2; |

اکنون تمام نمونه های Bird خاصیت numLegs را دارند.

|  |
| --- |
| console.log(duck.numLegs);  // prints 2  console.log(canary.numLegs);  // prints 2 |

از آنجا که همه نمونه ها به طور خودکار خصوصیات اولیه را دارند ، یک prototype را به عنوان "دستورالعمل" ایجاد اشیا در نظر بگیرید. توجه داشته باشید که prototype برای duck و canary بخشی از سازنده پرنده به عنوان Bird.prototype است. تقریباً هر شی در JavaScript دارای ویژگی prototype است که بخشی از عملکرد سازنده است که آن را ایجاد کرده است.

تمرین 7-10: یک ویژگی numLegs به نمونه اولیه Dog اضافه کنید

|  |
| --- |
| function Dog(name) {    this.name = name;  }  let beagle = new Dog("Snoopy"); |

## تکرار تمام خصوصیات

شما اکنون دو نوع ویژگی دیده اید: خواص اختصاصی و خواص نمونه اولیه. خصوصیات اختصاصی مستقیماً روی خود نمونه شی تعریف می شوند. و خصوصیات prototype بر روی prototype تعریف می شود.

|  |
| --- |
| function Bird(name) {    this.name = name;  //own property  }  Bird.prototype.numLegs = 2; // prototype property  let duck = new Bird("Donald"); |

در اینجا نحوه افزودن خصوصیات اختصاصی duck به آرایه ownProps و خصوصیات نمونه اولیه به آرایه prototypeProps وجود دارد:

|  |
| --- |
| let ownProps = [];  let prototypeProps = [];  for (let property in duck) {    if(duck.hasOwnProperty(property)) {      ownProps.push(property);    } else {      prototypeProps.push(property);    }  }  console.log(ownProps); // prints ["name"]  console.log(prototypeProps); // prints ["numLegs"] |

تمرین 7-11: تمام خصوصیات اختصاصی beagle را به آرایه ownProps اضافه کنید. تمام خصوصیات نمونه اولیه Dog را به آرایه prototypeProps اضافه کنید.

|  |
| --- |
| function Dog(name) {    this.name = name;  }  Dog.prototype.numLegs = 4;  let beagle = new Dog("Snoopy");  let ownProps = [];  let prototypeProps = []; |

## ویژگی Constructor را درک کنید

یک ویژگی خاص Constructorوجود دارد که درنمونه های شی duck و beagle قرار دارد که در چالش های قبلی ایجاد شده اند:

|  |
| --- |
| let duck = new Bird();  let beagle = new Dog();  console.log(duck.constructor === Bird);  //prints true  console.log(beagle.constructor === Dog);  //prints true |

توجه داشته باشید که ویژگی Constructor اشاره ای به تابع سازنده است که نمونه را ایجاد کرده است. مزیت ویژگی Constructor این است که می توان این ویژگی را بررسی کرد تا بفهمیم چه نوع شیئی است. در اینجا مثالی از نحوه استفاده از آن آورده شده است:

|  |
| --- |
| function joinBirdFraternity(candidate) {    if (candidate.constructor === Bird) {      return true;    } else {      return false;    }  } |

توجه داشته باشید

از آنجا که ویژگی constructor قابل رونویسی است (که در دو چالش بعدی به آن پرداخته می شود) به طور کلی بهتر است برای بررسی نوع یک شی از روش instanceof استفاده کنید.

تمرین 7-12: یک تابع joinDogFraternity بنویسید که یک پارامتر candidateرا بگیرد و با استفاده از ویژگی constructor ، اگر داوطلب یک Dog باشد ، true برگرداند و در غیر این صورت false باشد.

|  |
| --- |
| function Dog(name) {    this.name = name;  }  function joinDogFraternity(candidate) {  } |

## نمونه اولیه را به یک شی جدید تغییر دهید

تاکنون شما به صورت جداگانه خصوصیاتی را به نمونه اولیه اضافه کردید:

|  |
| --- |
| Bird.prototype.numLegs = 2; |

این پس از بیش از چند ویژگی خسته کننده می شود.

|  |
| --- |
| Bird.prototype.eat = function() {    console.log("nom nom nom");  }  Bird.prototype.describe = function() {    console.log("My name is " + this.name); |

یک روش کارآمدتر این است که نمونه اولیه را روی شی جدیدی تنظیم کنید که درحال حاضرحاوی خواصی باشد. به این ترتیب ، خصوصیات به یک باره اضافه می شوند:

|  |
| --- |
| Bird.prototype = {    numLegs: 2,    eat: function() {      console.log("nom nom nom");    },    describe: function() {      console.log("My name is " + this.name);    }  }; |

تمرین 7-13: خاصیت numLegs و دو متد eat() و describe() را با قرار دادن نمونه اولیه روی یک شی جدید به نمونه اولیه Dog اضافه کنید.

|  |
| --- |
| function Dog(name) {    this.name = name;  }  Dog.prototype = {}; |

## به یاد داشته باشید هنگام تغییر نمونه اولیه ، مشخصات سازنده را تنظیم کنید

تنظیم دستی نمونه اولیه روی یک شی جدید ، یک عارضه جانبی حیاتی دارد. این ویژگی سازنده را پاک می‌کند! از این ویژگی می توان برای بررسی اینکه کدام یک از توابع سازنده نمونه ایجاد کرده است استفاده کرد ، اما از آنجایی که این ویژگی بازنویسی شده است ، اکنون نتایج نادرستی به دست می دهد:

|  |
| --- |
| duck.constructor === Bird; // false -- Oops  duck.constructor === Object; // true, all objects inherit from Object.prototype  duck instanceof Bird; // true, still works |

برای رفع این مشکل ، هر زمان که نمونه اولیه به صورت دستی روی یک شی جدید تنظیم می شود ، به یاد داشته باشید که ویژگی سازنده را تعریف کنید:

|  |
| --- |
| Bird.prototype = {    constructor: Bird, // define the constructor property    numLegs: 2,    eat: function() {      console.log("nom nom nom");    },    describe: function() {      console.log("My name is " + this.name);    }  }; |

تمرین 7-14: ویژگی سازنده را در نمونه اولیه Dog تعریف کنید.

|  |
| --- |
| function Dog(name) {    this.name = name;  }  Dog.prototype = {    numLegs: 4,    eat: function() {      console.log("nom nom nom");    },    describe: function() {      console.log("My name is " + this.name);    }  }; |

## بدانید که نمونه اولیه یک شی از کجا می‌آید

درست مانند اینکه مردم ژن ها را از والدین خود به ارث می برند ، یک شی object نیز نمونه اولیه خود را مستقیماً از تابع سازنده ای که آن را ایجاد کرده به ارث می برد. به عنوان مثال ، در اینجا سازنده Bird شی Duck را ایجاد می کند:

|  |
| --- |
| function Bird(name) {    this.name = name;  }  let duck = new Bird("Donald"); |

duck نمونه اولیه خود را از تابع سازنده Bird به ارث می برد. می توانید این رابطه را با متد isPrototypeOf نشان دهید:

|  |
| --- |
| Bird.prototype.isPrototypeOf(duck);// returns true |

تمرین 7-15: برای بررسی نمونه اولیه beagle از isPrototypeOf استفاده کنید.

|  |
| --- |
| function Dog(name) {    this.name = name;  }  let beagle = new Dog("Snoopy"); |

## زنجیره نمونه اولیه را درک کنید

همه اشیا موجود در JavaScript (به استثنای چند مورد) دارای نمونه اولیه هستند. همچنین ، نمونه اولیه یک شی خود یک شی است.

|  |
| --- |
| function Bird(name) {    this.name = name;  }  typeof Bird.prototype; // yields 'object' |

از آنجا که نمونه اولیه یک شی است ، نمونه اولیه می تواند نمونه اولیه خود را داشته باشد! در این حالت ، نمونه اولیه Bird.prototype ، Object.prototype است:

|  |
| --- |
| Object.prototype.isPrototypeOf(Bird.prototype);   // returns true |

این چگونه مفید است؟ شما می توانید متد hasOwNProperty را از یک چالش قبلی به یاد بیاورید:

|  |
| --- |
| let duck = new Bird("Donald");  duck.hasOwnProperty("name"); // yields true |

متد hasOwnProperty در Object.prototype تعریف شده است که توسط Bird.prototype قابل دسترسی است ، که با اردک می توان به آن دسترسی پیدا کرد.

این نمونه ای از نمونه اولیه زنجیره ای است. در این زنجیره نمونه اولیه ، Bird ابرنوع duck است ، در حالی که duck زیرنوع است. شی هم برای Bird و هم برای duckیک ابرنوع است. شی برای تما اشیاء جاوااسکریپت یک ابرنوع است. بنابراین ، هر شی می تواند از متد hasOwnProperty استفاده کند.

تمرین 7-16: کد را تغییر دهید تا زنجیره نمونه اولیه صحیح نشان داده شود.

|  |
| --- |
| function Dog(name) {    this.name = name;  }  let beagle = new Dog("Snoopy");  Dog.prototype.isPrototypeOf(beagle);  // yields true  // Fix the code below so that it evaluates to true  ???.isPrototypeOf(Dog.prototype); |

## از ارث بری استفاده کنید تا دیگر خود را تکرار نکنید

در برنامه نویسی یک اصل وجود دارد به نام (خود را تکرار نکن)" “Don't Repeat Yourself (DRY). دلیل وجود کد تکرار شده به این دلیل است که هر تغییری نیاز به اصلاح کد در چندین مکان دارد. این معمولاً به معنای کار بیشتر برای برنامه نویسان و فضای بیشتر برای خطا است.

در مثال زیر توجه کنید که متد describe توسط Bird و Dog به اشتراک گذاشته شده است:

|  |
| --- |
| Bird.prototype = {    constructor: Bird,    describe: function() {      console.log("My name is " + this.name);    }  };  Dog.prototype = {    constructor: Dog,    describe: function() {      console.log("My name is " + this.name);    }  }; |

متد describe در دو مکان تکرار می شود. می توان کد را ویرایش کرد تا با ایجاد یک ابرنوع (یا والد) به نام Animal ، از اصل DRY پیروی کند:

|  |
| --- |
| function Animal() { };  Animal.prototype = {    constructor: Animal,    describe: function() {      console.log("My name is " + this.name);    }  }; |

از آنجا که Animal شامل متد describe است ، می توانید آن را از Bird و Dog حذف کنید:

|  |
| --- |
| Bird.prototype = {    constructor: Bird  };  Dog.prototype = {    constructor: Dog  }; |

تمرین 7-17: متد eat هم در Cat و هم در Bear تکرار می شود. با انتقال متد eat به ابرنوع Animal ، روح کد را با DRY ویرایش کنید.

|  |
| --- |
| function Cat(name) {    this.name = name;  }  Cat.prototype = {    constructor: Cat,    eat: function() {      console.log("nom nom nom");    }  };  function Bear(name) {    this.name = name;  }  Bear.prototype = {    constructor: Bear,    eat: function() {      console.log("nom nom nom");    }  };  function Animal() { }  Animal.prototype = {    constructor: Animal,  }; |

## رفتارهای خود را از یک ابرنوع به ارث ببرید

در چالش قبلی ، شما یک ابر نوع به نام Animal ایجاد کرده اید که رفتارهای مشترک همه حیوانات را تعریف می کند:

|  |
| --- |
| function Animal() { }  Animal.prototype.eat = function() {    console.log("nom nom nom");  }; |

این چالش و چالش بعدی شامل چگونگی استفاده مجدد از متدهای Animal در داخل Bird و Dog بدون تعریف مجدد آنها خواهد بود. از تکنیک به نام وراثت استفاده می کند. این چالش اولین قدم را شامل می شود: نمونه ای از ابر نوع (یا والد) ایجاد کنید. شما درحال حاضر یک روش برای ایجاد نمونه ای از Animal با استفاده از عملگر new می دانید:

|  |
| --- |
| let animal = new Animal(); |

هنگام استفاده از این نحو برای ارث بری ، معایبی وجود دارد که برای محدوه این چالش بسیار پیچیده است. در عوض ، در اینجا یک روش جایگزین بدون این معایب وجود دارد:

|  |
| --- |
| let animal = Object.create(Animal.prototype); |

Object.create(obj) یک شی جدید ایجاد می کند و obj را به عنوان نمونه اولیه شی جدید تعیین می‌کند. به یاد بیاورید که نمونه اولیه مانند "دستورالعمل" ایجاد یک شی است. با تنظیم نمونه اولیه animal به عنوان نمونه اولیه Animal's ، شما به طور موثر همان دستورالعمل animal را مانند هر نمونه دیگر از animal می دهید.

|  |
| --- |
| animal.eat(); // prints "nom nom nom"  animal instanceof Animal; // => true |

تمرین 7-18: برای ایجاد دو نمونه از Animal به نام duck و beagle از Object.create استفاده کنید.

|  |
| --- |
| function Animal() { }  Animal.prototype = {    constructor: Animal,    eat: function() {      console.log("nom nom nom");    }  };  let duck; // Change this line  let beagle; // Change this line |

## نمونه اولیه child را روی نمونه ای از Parent تنظیم کنید

در چالش قبلی ، اولین قدم برای به ارث بردن رفتار از ابرنوع (یا والدین) Animal را مشاهده کردید: ایجاد یک نمونه جدید از Animal.

این چالش گام بعدی را در بر می گیرد: تنظیم نمونه اولیه زیرنوع (یا child) - در این مورد ، پرنده - به عنوان نمونه ای از Animal.

|  |
| --- |
| Bird.prototype = Object.create(Animal.prototype); |

به یاد داشته باشید که نمونه اولیه مانند "دستورالعمل" ایجاد یک شی است. به نوعی ، دستورالعمل Bird اکنون شامل تمام "عناصر" اصلی Animal است.

|  |
| --- |
| let duck = new Bird("Donald");  duck.eat(); // prints "nom nom nom" |

duck تمام خصوصیات Animal از جمله متد eatرا به ارث می برد.

تمرین 7-19: کد را تغییر دهید تا نمونه هایی از Dog از Animal به ارث برسد.

|  |
| --- |
| function Animal() { }  Animal.prototype = {    constructor: Animal,    eat: function() {      console.log("nom nom nom");    }  };  function Dog() { }  let beagle = new Dog(); |

## تنظیم مجدد یک ویژگی سازنده به ارث برده شده

وقتی یک شی نمونه اولیه خود را از شی دیگری به ارث می برد ، ویژگی سازنده ابرنوع را نیز به ارث می برد. مثالی در اینجا آورده شده است:

|  |
| --- |
| function Bird() { }  Bird.prototype = Object.create(Animal.prototype);  let duck = new Bird();  duck.constructor // function Animal(){...} |

اما duck نمونه های دیگر Bird باید نشان دهند که آنها توسطBird ساخته شده اند نه Animal. برای انجام این کار ، می توانید خاصیت سازنده Bird را به صورت دستی روی شی Bird تنظیم کنید:

|  |
| --- |
| Bird.prototype.constructor = Bird;  duck.constructor // function Bird(){...} |

تمرین 7-20: کد را اصلاح کنید تا duck.constructor و beagle.constructor سازنده های مربوطه خود را بازگردانند.

|  |
| --- |
| function Animal() { }  function Bird() { }  function Dog() { }  Bird.prototype = Object.create(Animal.prototype);  Dog.prototype = Object.create(Animal.prototype);  let duck = new Bird();  let beagle = new Dog(); |

## بعد از وراثت متد ها را اضافه کنید

یک تابع سازنده که شیt نمونه اولیه خود را از یک تابع سازنده ابرنوع به ارث می برد ، می تواند علاوه بر متد های به ارث برده شده ، متد های خاص خود را نیز داشته باشد. به عنوان مثال ، Bird سازنده ای است که نمونه اولیه خود را از Animal به ارث می برد:

|  |
| --- |
| function Animal() { }  Animal.prototype.eat = function() {    console.log("nom nom nom");  };  function Bird() { }  Bird.prototype = Object.create(Animal.prototype);  Bird.prototype.constructor = Bird; |

علاوه بر آنچه از حیوانات به ارث رسیده است ، شما می خواهید رفتاری را اضافه کنید که مخصوص اشیا Bird است. در اینجا ، Bird تابع fly() را بدست می آورد. توابع به همان روش سایر توابع سازنده به نمونه اولیه پرنده اضافه می شوند:

|  |
| --- |
| Bird.prototype.fly = function() {    console.log("I'm flying!");  }; |

حال نمونه هایی از Bird هر دو متد eat() و fly() را خواهند داشت:

|  |
| --- |
| let duck = new Bird();  duck.eat(); // prints "nom nom nom"  duck.fly(); // prints "I'm flying!" |

تمرین 7-21: تمام کد لازم را اضافه کنید تا شی Dog از Animal به ارث برسد و سازنده نمونه اولیه Dog بر روی Dog تنظیم شود. سپس متد bark() را به شی Dog اضافه کنید تا beagle بتواند هم eat() و هم bark() داشته باشد. متد bark() باید "Woof!" را در کنسول چاپ کند.

|  |
| --- |
| function Animal() { }  Animal.prototype.eat = function() { console.log("nom nom nom"); };  function Dog() { }  let beagle = new Dog(); |

## متد های به ارث برده شده را بازنویسی بگیرید

در درس های قبلی ، شما یاد گرفتید که یک شی با ارجاع به نمونه اولیه خود می تواند رفتار (متد های) خود را از شی دیگری به ارث ببرد:

|  |
| --- |
| ChildObject.prototype = Object.create(ParentObject.prototype) |

سپس ChildObject متدهای خاص خود را با اتصال آنها روی نمونه اولیه خود دریافت کرد:

|  |
| --- |
| ChildObject.prototype.methodName = function() {...}; |

ممکن است یک متد به ارث برده شده را بازنویسی کنید. این کار به همان روش انجام می شود - با افزودن متدی به ChildObject.prototype با استفاده از همان نام متدی که بازنویسی می شود. در اینجا مثالی از Bird وجود دارد که متد eat() را از Animal به ارث برده است:

|  |
| --- |
| function Animal() { }  Animal.prototype.eat = function() {    return "nom nom nom";  };  function Bird() { }  // Inherit all methods from Animal  Bird.prototype = Object.create(Animal.prototype);  // Bird.eat() overrides Animal.eat()  Bird.prototype.eat = function() {    return "peck peck peck";  }; |

اگر نمونه ای مانند let duck = new Bird(); داریدو duck.eat() را فراخوانی می کنید ، JavaScript اینگونه به دنبال متد در زنجیره نمونه اولیه duck است:

1. duck => آیا eat() در اینجا تعریف شده است؟ نه.
2. Bird => آیا eat() در اینجا تعریف شده است؟ => بله. آن را اجرا کنید و جستجو را متوقف کنید.
3. Animal => eat() در اینجا نیز تعریف شده است ، اما JavaScript قبل از رسیدن به این سطح جستجو را متوقف کرد.
4. Object => جاوا اسکریپت قبل از رسیدن به این سطح جستجو را متوقف کرد.

تمرین 7-22: متد fly() را برای Penguin بازنویسی کنید تا "Alas, this is a flightless bird." را برگرداند.

|  |
| --- |
| function Bird() { }  Bird.prototype.fly = function() { return "I am flying!"; };  function Penguin() { }  Penguin.prototype = Object.create(Bird.prototype);  Penguin.prototype.constructor = Penguin;  let penguin = new Penguin();  console.log(penguin.fly()); |

## برای افزودن رفتار مشترک بین اشیا غیرمرتبط از Mixin استفاده کنید

همانطور که مشاهده کردید ، رفتار از طریق وراثت به اشتراک گذاشته می‌شود. با این حال مواردی وجود دارد که ارث بری درآن ها بهترین راه حل نیست. وراثت برای اشیا غیر مرتبط مانند پرنده و هواپیما به خوبی کار نمی کند. هر دو می توانند پرواز کنند ، اما پرنده نوعی هواپیما نیست و بالعکس.

برای اشیا غیر مرتبط بهتر است از mixin هااستفاده کنید. یک mixin به سایر اشیا اجازه می دهد تا از مجموعه ای از توابع استفاده کنند.

|  |
| --- |
| let flyMixin = function(obj) {    obj.fly = function() {      console.log("Flying, wooosh!");    }  }; |

flyMixin هر شی ای را می گیرد و متد fly (پرواز) را به آن می دهد.

|  |
| --- |
| let bird = {    name: "Donald",    numLegs: 2  };  let plane = {    model: "777",    numPassengers: 524  };  flyMixin(bird);  flyMixin(plane); |

در اینجا bird و plane به flyMixin ارسال می شوند ، که تابع fly را به هر شی ای اختصاص می دهد. اکنون پرنده (bird) و هواپیما plane)) هر دو می توانند پرواز کنند:

|  |
| --- |
| bird.fly(); // prints "Flying, wooosh!"  plane.fly(); // prints "Flying, wooosh!" |

توجه داشته باشید که چگونه mixin اجازه می دهد تا همان متد fly (پرواز) توسط اشیا غیر مرتبط bird (پرنده) و plane (هواپیما) استفاده شود.

تمرین 7-23: میکسینی به نام glide ایجاد کنید که متدی به نام glide را تعریف می کند. سپس از glideMixin استفاده کنید تا به پرنده(bird) و قایق(boat) توانایی سر خوردن(glide) را بدهید.

|  |
| --- |
| let bird = {    name: "Donald",    numLegs: 2  };  let boat = {    name: "Warrior",    type: "race-boat"  }; |

## از Closure برای محافظت از خصوصیات درون یک شی از اصلاح شدن خارجی استفاده کنید

در چالش قبلی ، bird یک خصوصیت عمومی به نام name داشت. عمومی در نظر گرفته می شود زیرا خارج از تعریف bird قابل دسترسی و تغییر است.

|  |
| --- |
| bird.name = "Duffy"; |

بنابراین ، هر قسمت از کد شما به راحتی می تواند نام پرنده را به هر مقداری تغییر دهد. به مواردی مانند گذرواژه و حساب بانکی فکر کنید که توسط هر بخشی از پایگاه کد شما به راحتی قابل تغییر هستند. این می‌تواند باعث بسیاری از مسائل شود.

ساده ترین راه خصوصی سازی این خاصیت عمومی ایجاد متغیر در تابع سازنده است. این دامنه آن متغیر را تغییر می دهد تا در تابع سازنده در مقابل سطح سراسری موجود باشد. به این ترتیب ، متغیر فقط با استفاده از متدهای موجود در تابع سازنده قابل دسترسی و تغییر است.

|  |
| --- |
| function Bird() {    let hatchedEgg = 10; // private variable    /\* publicly available method that a bird object can use \*/    this.getHatchedEggCount = function() {      return hatchedEgg;    };  }  let ducky = new Bird();  ducky.getHatchedEggCount(); // returns 10 |

در اینجا getHatchedEggCount یک روش ممتاز است ، زیرا به متغیر خصوصی hatchedEgg دسترسی دارد. این امکان وجود دارد زیرا hatchedEgg در همان بلوک به عنوان getHatchedEggCount اعلان می شود. در جاوا‌اسکریپت ، یک تابع همیشه به بلوکی که در آن ایجاد شده است دسترسی دارد. به این closure می‌گویند.

تمرین 7-24: نحوه اعلان weight در تابع Bird را تغییر دهید که یک متغیر خصوصی است. سپس یک متد getWeight ایجاد کنید که مقدار weight، 15 را برمی گرداند.

|  |
| --- |
| function Bird() {    this.weight = 15;  } |

## درک تابع بلافاصله اجرا یا فراخوانی شده(Immediately Invoked Function Expression) (IIFE)

الگوی رایج در JavaScript اجرای تابع به محض اعلان است:

|  |
| --- |
| (function () {    console.log("Chirp, chirp!");  })(); // this is an anonymous function expression that executes right away  // Outputs "Chirp, chirp!" immediately |

توجه داشته باشید که این تابع هیچ نامی ندارد و در یک متغیر ذخیره نمی شود. دو پرانتز () در انتهای عبارت تابع باعث می شود که بلافاصله اجرا یا فراخوانی شود. این الگو به عنوان تابع بلافاصله فراخوانی شده(Immediately Invoked Function Expression) (IIFE) شناخته می شود. این تابع در همان جا که اعلان می‌شود فراخوانی می‌شود.

تمرین 7-25: تابع makeNest را دوباره بنویسید و فراخوانی آن را حذف کنید درعوض آن یک عبارت تابع بلافاصله فراخوانی شده (IIFE) است.

|  |
| --- |
| function makeNest() {    console.log("A cozy nest is ready");  }  makeNest(); |

## برای ایجاد ماژول از IIFE استفاده کنید

یک تابع که بلافاصله فراخوانی می شود (IIFE) اغلب برای گروه بندی توابع مرتبط در یک شی یا ماژول استفاده می شود. به عنوان مثال ، چالش بعدی دو mixin را تعریف کرد:

|  |
| --- |
| function glideMixin(obj) {    obj.glide = function() {      console.log("Gliding on the water");    };  }  function flyMixin(obj) {    obj.fly = function() {      console.log("Flying, wooosh!");    };  } |

می توانیم این mixin ها را به صورت زیر در یک ماژول گروه بندی کنیم:

|  |
| --- |
| let motionModule = (function () {    return {      glideMixin: function(obj) {        obj.glide = function() {          console.log("Gliding on the water");        };      },      flyMixin: function(obj) {        obj.fly = function() {          console.log("Flying, wooosh!");        };      }    }  })(); // The two parentheses cause the function to be immediately invoked |

توجه داشته باشید که شما یک عبارت تابع بلافاصله فراخوانی شده دارید (IIFE) که یک motionModule را برمی‎گرداند. این شی برگشتی حاوی کلیه رفتارهای mixin به عنوان خصوصیات شی است. مزیت الگوی ماژول این است که تمام رفتارهای حرکتی را می توان در یک شی واحد بسته بندی کرد که می تواند توسط سایر قسمتهای کد شما استفاده شود. در اینجا مثالی با استفاده از آن آورده شده است:

|  |
| --- |
| motionModule.glideMixin(duck);  duck.glide(); |

تمرین 7-26: برای بسته بندی دو mixin، isCuteMixin و singMixin ماژولی به نام funModule ایجاد کنید. funModule باید یک شی را برگرداند.

|  |
| --- |
| let isCuteMixin = function(obj) {    obj.isCute = function() {      return true;    };  };  let singMixin = function(obj) {    obj.sing = function() {      console.log("Singing to an awesome tune");    };  }; |

# برنامه نویسی تابعی

## مقدمه ای بر چالش های برنامه نویسی کاربردی

برنامه نویسی تابعی رویکردی برای توسعه نرم افزار است که بر اساس ارزیابی توابع است. مانند ریاضیات ، توابع در برنامه نویسی ورودی و خروجی برای تولید نتیجه هستند. شما می توانید توابع اساسی را از بسیاری جهات برای ساخت برنامه های پیچیده و پیچیده تر ترکیب کنید.

برنامه نویسی کاربردی از چند اصل اساسی پیروی می کند:

* توابع از وضعیت برنامه یا متغیرهای سراری مستقل هستند. آنها فقط برای انجام محاسبه به آرگومان‌هایی که به داخلشان ارسال می‌شود بستگی دارند.
* توابع سعی می کنند هرگونه تغییر در وضعیت برنامه را محدود کرده و از تغییر در اشیا global (سراسری) نگهدارنده داده جلوگیری کنند.
* توابع حداقل عوارض جانبی را در برنامه دارند.

رویکرد توسعه نرم افزار برنامه نویسی کاربردی برنامه را به قطعات کوچک و قابل آزمایش تقسیم می کند. این بخش اصول اساسی برنامه نویسی کاربردی را در JavaScript پوشش می دهد.

## درباره برنامه نویسی تابعی بیاموزید

برنامه نویسی تابعی نوعی از برنامه نویسی است که در آن راه حل ها ساده، توابع ایزوله ، بدون هیچگونه عوارض جانبی خارج از محدوده تابع هستند.

INPUT -> PROCESS -> OUTPUT

برنامه نویسی تابعی در مورد موارد زیر است:

1. توابع ایزوله - هیچ وابستگی به وضعیت برنامه وجود ندارد ، که شامل متغیرهای جهانی است که قابل تغییر هستند.
2. توابع خالص - ورودی یکسان همیشه همان خروجی را می دهد
3. توابعی با عوارض جانبی محدود - هرگونه تغییر یا جهش در وضعیت برنامه خارج از تابع با دقت کنترل می شود.

تمرین 8-1: اعضای freeCodeCamp اتفاقاً عاشق چای هستند. در ویرایشگر کد ، توابع prepareTea و getTea قبلاً برای شما تعریف شده اند. با فراخوانی تابع getTea ، 40 فنجان چای برای تیم تهیه کنید و آنها را در متغیر tea4TeamFCC ذخیره کنید.

|  |
| --- |
| // تابعی که رشته ای را برمی‌گرداند که یک فنجان چای سبز را نشان می دهد  const prepareTea = () => 'greenTea';  /\*  با توجه به یک تابع (نشان دهنده نوع چای) و تعداد فنجان مورد نیاز ،  تابع زیر آرایه ای از رشته ها را برمی گرداند (هر کدام نشان دهنده یک فنجان از  نوع خاصی از چای است)/\*  const getTea = (numOfCups) => {    const teaCups = [];    for(let cups = 1; cups <= numOfCups; cups += 1) {      const teaCup = prepareTea();      teaCups.push(teaCup);    }    return teaCups;  };  // فقط کد زیر را تعقی دهید  const tea4TeamFCC = null; |

## اصطلاحات برنامه نویسی تابعی را بشناسید

تیم FCC نوسانات روحی داشت و اکنون دو نوع چای می خواهد: چای سبز و چای سیاه. واقعیت عمومی: تغییرات خلقی مشتری بسیار رایج است.

با استفاده از این اطلاعات ، برای رسیدگی به درخواست های مختلف چای ، باید دوباره نگاهی به آخرین چالش تابع getTea بیندازیم. ما می توانیم getTea را تغییر دهیم تا تابعی را به عنوان یک پارامتر بپذیرد تا بتواند نوع چای تهیه شده را تغییر دهد. این باعث می شود getTea انعطاف پذیرتر باشد و در هنگام تغییر درخواست مشتری ، کنترل بیشتری به برنامه نویس می دهد.

اما ابتدا ، بیایید برخی اصطلاحات تابعی را پوشش دهیم:

Callbackها توابعی هستند که برای تصمیم گیری در مورد فراخوانی آن تابع لغزیده یا به تابع دیگری ارسال می‌شود. ممکن است دیده باشید که آنها به متد‌های دیگری ارسال شده اند ، به عنوان مثال در filter، تابع callback معیارهای نحوه فیلتر کردن آرایه را به JavaScript می گوید.

توابعی که می توانند به یک متغیر اختصاص داده شوند ، به یک تابع دیگر ارسال شوند ، یا مانند هر مقدار عادی دیگر از تابع دیگری برگردانده شوند ، توابع درجه اول نامیده می شوند. در جاوا اسکریپت ، همه توابع، توابع درجه اول هستند.

به توابعی که تابعی را به عنوان آرگومان می گیرند یا تابعی را به عنوان مقدار برگشتی برمی گردانند ، توابع مرتبه بالاتر می گویند.

وقتی توابع به تابع دیگری ارسال می شوند یا از تابع دیگری برمی گردند ، آنگاه این توابع را که ارسال می شوند یا برمی گردانند می توان lambda نامید.

تمرین 8-1: 27 فنجان چای سبز و 13 فنجان چای سیاه آماده کرده و به ترتیب در متغیرهای tea4GreenTeamFCC و tea4BlackTeamFCC ذخیره کنید. توجه داشته باشید که تابع getTea اصلاح شده است بنابراین اکنون یک تابع را به عنوان اولین آرگومان می گیرد.

توجه: داده ها (تعداد فنجان چای) به عنوان آخرین آرگومان ارائه می شوند. در درس های بعدی بیشتر در این مورد بحث خواهیم کرد.

|  |
| --- |
| // تابعی که رشته ای را برمی‌گرداند یک فنجان چای سبز را نشان می دهد  const prepareTea = () => 'greenTea';  // تابعی که رشته ای را برمی گرداند یک فنجان چای سیاه را نشان می دهد  const prepareBlackTea = () => 'blackTea';  /\*  با توجه به یک تابع (نشان دهنده نوع چای) و تعداد فنجان مورد نیاز ،  تابع زیر آرایه ای از رشته ها را برمی گرداند (هر کدام نشان دهنده یک فنجان از  نوع خاصی از چای است)/\*  const getTea = (prepareTea, numOfCups) => {    const teaCups = [];    for(let cups = 1; cups <= numOfCups; cups += 1) {      const teaCup = prepareTea();      teaCups.push(teaCup);    }    return teaCups;  };  // فقط کد زیر را تعقیر دهید  const tea4GreenTeamFCC = null;  const tea4BlackTeamFCC = null;  // فقط کد بالا را تعقیر دهید  console.log(    tea4GreenTeamFCC,    tea4BlackTeamFCC  ); |

## خطرات استفاده از کد imperative را درک کنید

برنامه نویسی تابعی عادت خوبی است. این مدریت کد شما را آسان کنترل می کند و شما را از اشکالات اب زیر کاه نجات می دهد. اما قبل از اینکه به آنجا برویم ، بیایید یک رویکرد imperative برای برنامه نویسی را بررسی کنیم تا مشخص کنیم که در چه مواردی ممکن است با مشکل روبرو شوید.

در انگلیسی (و بسیاری از زبان‌های دیگر) ، از زمان imperative برای دستور دادن استفاده می شود. به همین ترتیب ، یک سبک imperative در برنامه نویسی نوعی است که به کامپیوتر مجموعه ای از دستورات را برای انجام یک کار می دهد.

اغلب دستورات مانند به روزرسانی متغیرهای سراسری ، وضعیت برنامه را تغییر می دهند. یک مثال کلاسیک نوشتن یک حلقه for است که جهت دقیق تکرار بیش از ایندکس های یک آرایه را می دهد.

در مقابل ، برنامه نویسی تابعی نوعی برنامه نویسی اعلانی است. با فراخوانی یک متد یا تابع ، به کامپیوتر می‌گویید که می خواهید چه کاری انجام شود. جاوا اسکریپت متدای از پیش تعریف شده بسیاری را ارائه می‌دهد که وظایف معمول را بر عهده دارند بنابراین نیازی به نوشتن نحوه انجام آنها توسط رایانه نیست.

به عنوان مثال ، به جای استفاده از حلقه for که در بالا ذکر شد ، می توانید متد map را فراخوانی کنید که جزئیات تکرار در یک آرایه را کنترل می کند. این امر به جلوگیری از خطاهای معنایی کمک می کند ، مانند "Off by One Errors" که در بخش اشکال زدایی پوشش داده شد.

این سناریو را در نظر بگیرید: شما در حال مرور وب در مرورگر خود هستید و می خواهید برگه هایی را که باز کرده اید دنبال کنید. بیایید سعی کنیم این مدل را با استفاده از چند کد ساده شی گرا مدل کنیم.

یک شی Window از زبانه ها تشکیل شده است و شما معمولاً بیش از یک پنجره باز دارید. عناوین هر سایت باز در هر شی Window به صورت آرایه ای نگهداری می شود. پس از کار در مرورگر (باز کردن برگه های جدید ، ادغام پنجره ها و بستن برگه ها) ، می خواهید برگه هایی را که هنوز باز هستند را چاپ کنید. برگه های بسته از آرایه حذف می‌شوند و برگه های جدید (برای سادگی) به انتهای آن اضافه می شوند.

ویرایشگر کد اجرای این قابلیت تابعی را با توابع tabOpen () ، tabClose() و join() نشان می دهد. زبانه های آرایه بخشی از شی Window است که نام صفحات باز شده را ذخیره می کند.

تمرین 8-3: کد را در ویرایشگر بررسی کنید. این از متدی استفاده می کند که دارای عوارض جانبی در برنامه است و باعث رفتار نادرست می شود. لیست نهایی برگه های باز ، ذخیره شده در finalTabs.tabs ، باید اینگونه باشد:

['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium', 'new tab', 'Netflix', 'YouTube', 'Vine', 'GMail', 'Work mail', 'Docs', 'freeCodeCamp', 'new tab']

اما لیست تولید شده توسط کد کمی متفاوت است. Window.prototype .tabClose را تعقیر دهید تا برگه درست را حذف کند.

|  |
| --- |
| tab//ها  آرایه ای از عناوین هر سایت است که در پنجره باز می شود  var Window = function(tabs) {    this.tabs = tabs; // ما یک رکورد از آرایه را داخل شی نگه می داریم};  // وقتی دو پنجره را به یک پنجره متصل می کنید  Window.prototype.join = function (otherWindow) {    this.tabs = this.tabs.concat(otherWindow.tabs);    return this;  };  // وقتی در انتها برگه جدیدی را باز می کنید  Window.prototype.tabOpen = function (tab) {    this.tabs.push('new tab'); // اکنون یک برگه جدید باز می کنیم    return this;  };  // وقتی یک برگه را می بندید  Window.prototype.tabClose = function (index) {    // فقط کد زیر این خط را تعقیر دهید    var tabsBeforeIndex = this.tabs.splice(0, index); // Get the tabs before the tab    var tabsAfterIndex = this.tabs.splice(index + 1); // Get the tabs after the tab    this.tabs = tabsBeforeIndex.concat(tabsAfterIndex); // Join them together    // فقط کد بالای این خط را تعقیر دهید    return this;   };  // بیایید سه پنجره مرورگر ایجاد کنیم  var workWindow = new Window(['GMail', 'Inbox', 'Work mail', 'Docs', 'freeCodeCamp']); // Your mailbox, drive, and other work sites  var socialWindow = new Window(['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium']); // Social sites  var videoWindow = new Window(['Netflix', 'YouTube', 'Vimeo', 'Vine']); // Entertainment sites  // اکنون باز کردن ، بستن و سایر عملیات برگه ها را اجرا کنید  var finalTabs = socialWindow    .tabOpen() // Open a new tab for cat memes    .join(videoWindow.tabClose(2)) // Close third tab in video window, and join    .join(workWindow.tabClose(1).tabOpen());  console.log(finalTabs.tabs); |

## با استفاده از برنامه نویسی تابعی از جهش و عوارض جانبی خودداری کنید

اگر قبلاً متوجه نشده اید ، مسئله در چالش قبلی مربوط به فراخوانی splice در تابع tabClose() بوده است. متأسفانه ، splice آرایه اصلی را که فراخوانی می شود تغییر می دهد ، بنابراین در فراخوانی دوم از یک آرایه اصلاح شده استفاده شد و نتایج غیر منتظره ای را به همراه داشت.

این یک مثال کوچک از یک الگوی بسیار بزرگتر است - شما یک تابع را روی یک متغیر ، آرایه یا یک شی فراخوانی می کنید ، و تابع متغیر یا چیزی را در شی تغییر می دهد.

یکی از اصول اصلی برنامه نویسی تابعی تغییر ندادن چیزها هست. تغییرات منجر به اشکال می شود. با دانستن اینکه توابع شما چیزی را تغییر نمی دهد جلوگیری از اشکالات ساده تر است ، از جمله آرگومان های تابع یا هر متغیر سراسری.

مثال قبلی هیچ عملیات پیچیده ای نداشت اما متد splice آرایه اصلی را تغییر داد و منجر به اشکال شد. به یاد بیاورید که در برنامه نویسی تابعی ، تغییر یا اصلاح چیزها جهش ، و نتیجه آن عارضه جانبی(side effect) نامیده می شود.

یک تابع ، در حالت ایده آل ، باید یک تابع خالص(pure) باشد ، به این معنی که هیچ عارضه جانبی ایجاد نمی‌کند. بیایید سعی کنیم بر این نظم تسلط پیدا کنیم و هیچ متغیر یا شیئی را در کد خود تغییر ندهیم.

تمرین 8-4: کد تابع incrementerرا پر کنید تا مقدار متغیر fixedValue سراسری را که یک واحد افزایش پیدا می‌کند را برگرداند.

|  |
| --- |
| // متغیر سراسری  var fixedValue = 4;  function incrementer () {    // Only change code below this line    // Only change code above this line  } |

## برای جلوگیری از وابستگی خارجی به یک تابع ، آرگومان‌هایی را ارسال کنید

آخرین چالش یک قدم نزدیک به اصول برنامه نویسی تابعی بود ، اما هنوز چیزی وجود ندارد. ما مقدار متغیر سراسری را تغییر ندادیم ، اما تابع incrementer بدون وجود متغیرسراسری fixedValue کار نمی کند.

یکی دیگر از اصول برنامه نویسی کاربردی این است که همیشه وابستگی های خود را صریحاً اعلان کنید. این بدان معناست که اگر یک تابع به یک متغیر یا شی موجود بستگی دارد ، سپس آن متغیر یا شی را مستقیماً به عنوان آرگومان به آن تابع ارسال کنید.

چندین نتیجه خوب از این اصل وجود دارد. تست تابع آسان تر است ، شما دقیقاً می دانید که چه ورودی می‌گیرد و این به هیچ چیز دیگری در برنامه شما بستگی نخواهد داشت.

با تغییر ، حذف یا افزودن کد جدید ، این امر می تواند به شما اطمینان بیشتری دهد. شما می دانید که چه چیزی را می توانید تغییر دهید یا تغییر ندهید و می توانید دام های احتمالی را ببینید.

سرانجام ، تابع همیشه خروجی یکسانی را برای همان مجموعه ورودی تولید می کند ، مهم نیست که کدام قسمت از کد آن را اجرا می کند.

تمرین 8-5: بیایید تابع incrementerرا به روز کنیم تا وابستگی های آن را به وضوح اعلان کنیم. تابع incrementer را بنویسید تا یک آرگومان را بگیرد ، و پس از افزایش مقدار یک به یک نتیجه را برمی گرداند.

|  |
| --- |
| // متغیر سراسری  var fixedValue = 4;  // Only change code below this line  function incrementer () {    // Only change code above this line  } |